NASA/CR-1999-209827
ICASE Report No. 99-50

Efficient Symbolic State-space Construction for Asynchronous Systems

Gianfranco Ciardo
The College of William & Mary, Williamsburg, Virginia

Gerald Lüttgen
ICASE, Hampton, Virginia

Radu Siminiceanu
The College of William & Mary, Williamsburg, Virginia

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA
Operated by Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NAS1-97046

December 1999

EFFICIENT SYMBOLIC STATE-SPACE CONSTRUCTION FOR ASYNCHRONOUS SYSTEMS*

GIANFRANCO CIARDO†, GERALD LÜTTGEN‡, AND RADU SIMINICEANU†

Abstract. Many state-of-the-art techniques for the verification of today's complex embedded systems rely on the analysis of their reachable state spaces. In this paper, we develop a new algorithm for the symbolic generation of the state space of *asynchronous* system models, such as Petri nets. The algorithm is based on previous work that employs *Multi-valued Decision Diagrams* (MDDs) for efficiently storing sets of reachable states. In contrast to related approaches, however, it fully exploits *event locality* which is a fundamental semantic property of asynchronous systems. Additionally, the algorithm supports intelligent *cache management* and achieves faster convergence via advanced *iteration control.* It is implemented in the tool SMART, and run-time results for several examples taken from the Petri net literature show that the algorithm performs about one order of magnitude faster than the best existing state-space generators.

Key words. event locality, multi-valued decision diagrams, state-space exploration

Subject classification. Computer Science

1. Introduction. The high complexity of today's embedded systems requires the application of rigorous mathematical techniques to testify to their proper behavior. Many of these techniques, including model checking [8], rely on the *automated construction of the reachable state space* of the system under consideration. However, state spaces of real-world systems are usually very large, sometimes too large to fit in a workstation's memory. One contributing factor to this problem is the concurrency inherent in many embedded systems, such as specified by *Petri nets* [20]. In fact, the size of the state space of an asynchronous, concurrent system is potentially exponential in the number of its parallel components. Consequently, many research efforts in *state-exploration techniques* have concentrated on the efficient exploration and storage of very large state spaces. In the literature, two principal research directions are considered, which differ from each other by whether sets of states are stored *explicitly* or *symbolically.*

Explicit techniques represent the reachable state spaces of systems by trees, hash tables, or graphs, where each state corresponds to an entity of the underlying data structure [3, 6, 10, 13]. Thus, the memory needed to store the state space of a system is linear in the number of the system's states, which in practice limits these techniques to fairly small systems having at most a few million states. However, since state spaces are encoded explicitly in their natural form, *minimization techniques* with respect to behavioral equivalences [12] or *partial-order techniques* [11] may be applied to reduce the sizes of state spaces further. Explicit techniques prove especially advantageous if one is interested in the numerical analysis of Markov processes defined over such state spaces [16].

*This work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-2199, USA.

†Department of Computer Science, P.O. Box 8795, College of William and Mary, Williamsburg, VA 23187-8795, USA, email: {ciardo, radu}@cs.wm.edu.

‡ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681-2199, USA, e-mail: luettgen@icase.edu.

Symbolic techniques allow one to store reachability sets in sublinear space. Most symbolic approaches use *Binary Decision Diagrams* (BDDs) as data structure for efficiently representing Boolean functions [1], into which state spaces can be mapped. The advent of BDD-based techniques pushed the manageable sizes of state spaces to about 10^{20} states [4]. In the Petri net community, BDDs were first applied by Pastor et al. [22] for the generation of the reachability sets of *safe* Petri nets and, subsequently, efficient encodings for other classes of Petri nets into BDDs were investigated [21]. Recently, symbolic state-space generation for Petri nets has been significantly improved [19]. The approach taken in [19] does not rely on BDDs, but is based on the more general concept of *Multi-valued Decision Diagrams* (MDDs) [15]. MDDs essentially represent integer functions and allow one to efficiently encode the state of an entire subnet of a Petri net using only a single integer variable, where the state spaces of the subnets are built by employing traditional techniques. Experimental results reported in [19] show that this approach enables the representation of even larger state spaces of size 10^{60} and even 10^{600} states for particularly regular nets. However, the time needed to generate some of these state spaces ranges from several minutes for the *dining philosophers* [22], with 1000 philosophers, to several hours for the Kanban system [6], with an initial token count of 75 tokens. Thus, while symbolic techniques are able to store larger and larger state spaces, state-space generation shifts from a *memory-bound* to a *time-bound* problem.

The objective of this paper is to improve on the time efficiency of symbolic state-space generation techniques for a particular class of systems, namely *asynchronous systems*. This class is especially interesting since it includes many embedded software systems. Our approach exploits the concept of *event locality*, or *interleaving*, inherent in asynchronous systems. In Petri nets, for example, event locality means that only those sub-markings belonging to the subnets affected by a given transition need to be updated when the transition fires. Whereas event locality has been investigated in explicit state-space generation techniques [5], it has been largely ignored in symbolic techniques. Only the MDD-based approach presented in [19] touches on event locality, but it exploits this concept only superficially. In particular, this approach does not support direct jumps to the part of the MDD corresponding to the submarkings that need to be updated when a transition fires. Similarly, it does not consider jumping out of the MDD upon finishing a local update of the data structure. The present paper develops a new algorithm for building the reachable state spaces of asynchronous systems, which is based on the algorithm described in [19]. Like [19], it uses MDDs for representing state spaces; unlike [19], it fully exploits event locality. Moreover, it introduces an intelligent mechanism for *cache management*, and also achieves faster convergence by firing events in a specific, predefined order. The new algorithm is implemented in the tool SMART [5] and is applied to explore the reachable state spaces of a suite of well-known Petri net models. It turns out that this algorithm is about one order of magnitude faster than the one presented in [19]. Remarkably, this improvement is mainly achieved by exploiting event locality and induces only a small overhead regarding space efficiency.

The remainder of this paper is organized as follows. The next section provides some background material regarding structured state spaces and MDDs. Secs. 3 and 4 focus on several conceptual issues, based on the notion of event locality, which are essential for deriving our new MDD-based algorithm in two variants. Details of the variants are presented in Sec. 5, while Sec. 6 discusses some performance results. Secs. 7 and 8 refer to related work and present our conclusions as well as directions for future work, respectively. Finally, Appendices A–C contain the detailed pseudo code of our algorithm, while Appendix D illustrates the algorithm step-by-step for a small example system.

2. Structured State Spaces and Multi-valued Decision Diagrams. This section gives a brief introduction and defines some notation regarding structured state spaces and multi-valued decision diagrams.

2.1. Structured State Spaces. We choose to specify finite-state asynchronous systems by Petri nets, noting that, however, the concepts and techniques presented in this paper are not limited to this choice. Thus, we interchangeably use the notions *net* and *system*, *subnet* and *sub-system*, *transition* and *event*, *marking* and *(global) state*, as well as *sub-marking* and *local state*.

Consider a Petri net with a finite set $\mathcal{P}$ of places, a finite set $\mathcal{E}$ of events, and an initial marking $s_0 \in \mathbb{N}^{|\mathcal{P}|}$. The interleaving semantics of Petri nets [20] defines how the *firing* of an event e can move the net from some state s to another state s'. We denote the set of successor states, or "next states," which are reachable from state s via event e by $\mathcal{N}(e,s)$. If $\mathcal{N}(e,s) = \emptyset$, event e is *disabled* in s; otherwise, it is *enabled*. For Petri nets, $\mathcal{N}$ is essentially a simple encoding of the input and output arcs; thus, $\mathcal{N}(e,s)$ contains at most one element. For other formalisms, however, $\mathcal{N}(e,s)$ might contain several elements. We are interested in exploring the set $\mathcal{S}$ of reachable states of the net under consideration. $\mathcal{S}$ is formally defined as the smallest set that (i) contains the initial state s_0 of the net and (ii) is closed under the "one-step reachability relation," i.e., if $s \in \mathcal{S}$, then $\mathcal{N}(e,s) \subseteq \mathcal{S}$, for any event e defined in the net.

As in [19], our encoding of the state space of a Petri net requires us to partition the net into K subnets by splitting its set of places $\mathcal{P}$ into K subsets $\mathcal{P}_K, \mathcal{P}_{K-1}, \ldots, \mathcal{P}_1$. This implies a partition of a *global state* s of the net into K *local states*, i.e., s has the form $(s_K, s_{K-1}, \ldots, s_1)$. The partition of $\mathcal{P}$ must satisfy a fundamental *product-form requirement*, which is also needed in Kronecker approaches for computing the solution of structured Markov models [7]. The product form demands for function $\mathcal{N}$ to be written as the cross-product of K local next-state functions, i.e., $\mathcal{N}(e,s) = \mathcal{N}_K(e,s_K) \times \mathcal{N}_{K-1}(e,s_{K-1}) \times \cdots \times \mathcal{N}_1(e,s_1)$ for all $e \in \mathcal{E}$ and $s \in \mathcal{S}$. Furthermore, in practice, each subnet should be small enough such that its reachable *local state space* $\mathcal{S}_k = \{s_{k,0}, s_{k,1}, \ldots, s_{k,N_k-1}\}$ can be efficiently computed by traditional techniques, where $N_k \in \mathbb{N}$ is the number of reachable states in subnet k. Note that this might require the explicit insertion of additional constraints, for example expressed through implicit places, to allow for the correct computation of $\mathcal{S}_k$ in isolation. In reality, one may use a small superset of the "true" $\mathcal{S}_k$, e.g., obtained by employing *p-invariants* [20]. Once $\mathcal{S}_k$ has been built, we can identify it with the set $\{0, 1, \ldots, N_k - 1\}$. Moreover, a set $\mathcal{S}$ of global states can then be encoded by the *characteristic function*

$$f_{\mathcal{S}} : \{0, \ldots, N_K - 1\} \times \{0, \ldots, N_{K-1} - 1\} \times \cdots \times \{0, \ldots, N_1 - 1\} \longrightarrow \{0, 1\}$$

defined by $f(s_K, s_{K-1}, \ldots, s_1) = 1$ if and only if $(s_K, s_{K-1}, \ldots, s_1) \in \mathcal{S}$. Such characteristic functions can be stored and manipulated efficiently, as suggested in the following sections.

2.2. Multi-valued Decision Diagrams. *Multi-valued Decision Diagrams* [15], or MDDs for short, are data structures for efficiently representing integer functions of the form

$$f : \{0, \ldots, N_K - 1\} \times \{0, \ldots, N_{K-1} - 1\} \times \cdots \times \{0, \ldots, N_1 - 1\} \longrightarrow \{0, \ldots, M - 1\}$$

where $K, M \in \mathbb{N}$ and $N_k \in \mathbb{N}$, for $K \geq k \geq 1$. When $M = 2$ and $N_k = 2$, for $K \geq k \geq 1$, function f is a Boolean function, and MDDs coincide with the better known *Binary Decision Diagrams* (BDDs) [1, 2]. Another special case, where $M = 2$, are the characteristic functions mentioned in the previous section.

Traditionally, integer functions are often represented by *value tables* or *decision trees*. Fig. 2.1, left-hand side, shows the decision tree of the minimum function *min(a,b,c)*, where the variables a, b, and c are taken

from the set $\{0,1,2\}$. Hence, $K = 3$ and $N_1 = N_2 = N_3 = M = 3$. Each internal node, which is depicted by an oval, is labeled by a variable and has arcs directed towards its three children. The i-th branch corresponds to the case where the variable of the node under consideration is assigned value i. Moreover, all nodes at a given level of the tree are labeled by the same variable, i.e., all paths through the tree have the same *variable ordering*, which in our example is $a < b < c$. Leaf nodes, depicted by squares, are labeled by either 0, 1, or 2. Each path from the root to a leaf node corresponds to an assignment of the variables to values. The value of the leaf in a given path is the value of the function with respect to the assignment for this path.

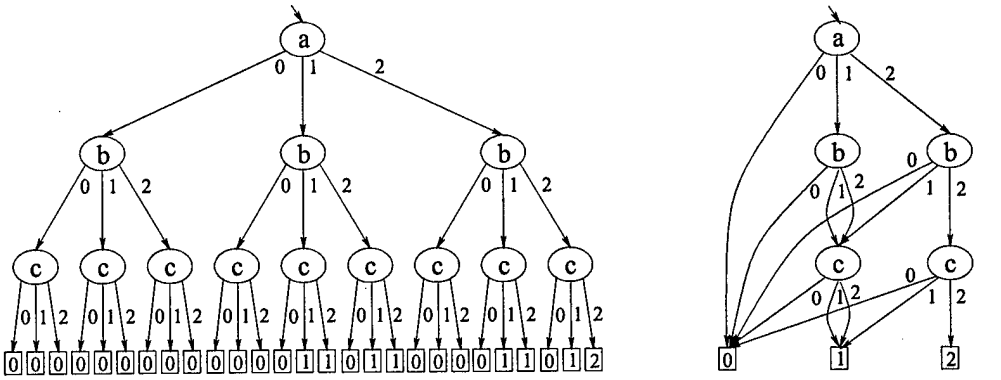

FIG. 2.1. *Representation of min(a,b,c) as decision tree (left) and as MDD (right)*

An MDD is a representation of a decision tree as *directed acyclic graph*, where identical subtrees are merged. More precisely, MDDs are *reduced* decision trees which do not contain any *non-unique* or *redundant* node. A node is considered to be non-unique if it is a replica of another node, and to be redundant if all its children are identical. Together with a fixed variable ordering, these two requirements ensure that MDDs provide a *canonical* representation of integer functions [15]. Note that the elimination of redundant nodes implies that arcs can skip levels. For example, the arc labeled with 0 connecting node a to leaf node 0 in Fig. 2.1, right-hand side, skips levels b and c. This means that the value of the function is 0, whenever a is 0. MDD representations can be exponentially more compact than their corresponding value tables or decision trees. However, the degree of compactness depends on the chosen variable ordering.

2.3. Data Structures for MDDs. We organize MDD nodes in levels ranging from K, at the top, to 1, at the bottom. Additionally, there is the special level 0, which contains either or both leaf nodes corresponding to the values 0 and 1, indicating whether a state is reachable or not. In practice, however, there is no need to store these nodes explicitly. The addresses of the nodes at a given level are stored within a hash table, to provide fast access to them and to simplify detection of non-unique nodes. Hence, we have K hash tables which together represent an MDD. We also refer to this data structure as *unique table*. Each node at level k consists of an array of N_k node addresses, which contain the arcs to the children of the node. Since we enforce the reducedness property, we use the value of this array to compute the hash value of the node. In the following, we let *mddNode* denote the type of nodes and *mddAddr* the type of addresses of nodes. Note that we could also use a single unique table for representing MDDs, but this would require us to store the level of a node as part of *mddNode*; furthermore, the level-wise organization of our data structures will prove very useful for the purposes of this paper. For notational simplicity, we often write $\langle lvl, ind \rangle$ for the node q stored in the *lvl*-th unique table at position *ind*, and $q{\rightarrow}dw[i]$ for the i-th child of q. Finally, we use nodes $\langle 0,0 \rangle$ and $\langle 0,1 \rangle$ to indicate the Boolean values 0 and 1 at level 0, respectively.

TABLE 2.1
"Union" operation on MDDs

```
Union(in p : mddAddr, in q : mddAddr) : mddAddr
 1. if p = ⟨0,1⟩ or q = ⟨0,1⟩ return ⟨0,1⟩;          • deal with the base cases first
 2. if p = ⟨0,0⟩ or p = q return q;
 3. if q = ⟨0,0⟩ return p;
 4. k ⇐ Max(p.lvl, q.lvl);                          • maximum of the levels of p and q
 5. if LookUpInUC(k,p,q,r) then return r;           • if found in the union cache, the result is returned in r
 6. r ⇐ CreateNode(k);                              • otherwise, the union needs to be computed in r
 7. for i = 0 to N_k − 1 do                         • for the i-th child do...
 8.     if k > p.lvl then u ⇐ Union(p, q→dw[i]);    • p is at a lower level than q
 9.     else if k > q.lvl then u ⇐ Union(p→dw[i], q); • q is at a lower level than p
10.     else u ⇐ Union(p→dw[i], q→dw[i]);           • p and q are at the same level
11.     SetArc(r,i,u);                             • make u the i-th child of r
12. r ⇐ CheckNode(r);                              • if r is unique and non-redundant, store it in the unique table
13. InsertInUC(k,p,q,r);                           • record the result of this union in the union cache
14. return r;                                      • return MDD representing the "union" of p and q
```

2.4. The *Union* Operation on MDDs. An essential operation for generating reachable state spaces is the binary *union* on sets. Since in our context all sets are represented as MDDs, an algorithm is needed which takes two MDDs as parameters and returns a new MDD, representing the union of the sets represented by its arguments. This algorithm, which is very similar to the one used in [19], that in turn is adapted from a BDD-based algorithm [2], is shown in Table 2.1. It recursively analyzes the argument MDDs, when descending from the maximum level k of the argument MDDs to the lowest level 0, and builds the result MDD, when finishing the recursions by ascending from level 0 to level k. Note that the maximum of the levels of the argument MDDs is the highest level the result MDD can have.

The base cases of the recursive function *Union* are handled in Lines 1–3, where the MDDs $\langle 0,0\rangle$ and $\langle 0,1\rangle$ encode the *empty set* and the *full set*, respectively. If $k > 0$, a *union cache* is used to check whether the union of the arguments p and q has been computed previously. If so, the result stored in the cache is returned. Otherwise, a new MDD node at level k is created whose i-th child is determined by recursively building the union of the i-th child of p and the i-th child of q, for all $0 \le i \le N_k$ (cf. Lines 7–11). However, one needs to take care of the fact that some child might not be explicitly represented, namely if it is redundant (cf. Lines 8 and 9). Finally, to ensure that the resulting MDD is reduced, node r is checked by calling function *CheckNode*(r). If r is redundant, then *CheckNode* destroys r and returns r's child, and if r is equivalent to another node r' having the same children, then *CheckNode* destroys r and returns r'. Otherwise, *CheckNode* inserts node r in the unique table and returns it. Note that the algorithm in Table 2.1 can be easily adapted for computing many other binary operations, such as intersection, by modifying Lines 1–3 accordingly.

2.5. MDD-based State-space Construction. Table 2.2 shows a naive, iterative, and MDD-based algorithm to build the reachable state space of a system represented by a Petri net. As explained earlier, the state space is encoded as a characteristic function, so a global state $s = (s_K, s_{K-1}, \ldots, s_1)$ is stored over the K levels of the MDD, one substate per level. Please recall that this requires us to partition Petri nets into subnets. While this can in principle be done automatically, it is still an open problem how to efficiently find "good" partitions, i.e., those that lead to small MDD representations of reachable state spaces. We refer the reader to [19] for a detailed discussion of issues regarding partitioning.

TABLE 2.2
Iterative state-space generation

```
MDDgeneration(in m : array[1,...,K] of int) : mddAddr
 1. for k = 1 to K do ClearUT(k);          • clear unique table
 2. q ⇐ SetInitial(m);                     • build and return MDD representing the initial state
 3. repeat                                 • start state-space exploration
 4.     for k = 1 to K do ClearUC(k);      • clear union cache
 5.     for k = 1 to K do ClearFC(k);      • clear firing cache
 6.     mddChanged ⇐ false;                • true if MDD changes in this iteration
 7.     foreach event e do Fire(e, q, mddChanged)   • fire event e and add newly reached states to MDD
 8. until mddChanged = false;              • keep iterating until fixed point is reached
 9. return q;                              • return MDD representing the reachable state space
```

The semantics of the Petri net under study is encoded in procedure *Fire* (cf. Table 2.2), which updates the MDD rooted at q according to the firing of event e by appropriately applying the *Union* operation shown above. For efficiency reasons, it also makes use of another cache, which we refer to as *firing cache*. The procedure additionally updates a flag *mddChanged*, if the firing of e added any new reachable states. After first clearing the unique table, the initial marking m of the Petri net under consideration is stored as an MDD via procedure *SetInitial*. The algorithm then proceeds iteratively. In each iteration, every enabled Petri net transition is fired, and the potentially new states are added to the MDD. This is done until the MDD does not change, i.e., until no more reachable states are discovered. Finally, the root node q, representing the reachable state space of the Petri net, is returned.

3. The Concept of Event Locality. Our improvements for the MDD-based generation of reachable state spaces rely on the notion of event locality, which asynchronous systems inherently obey.

Event locality, which is sometimes also referred to as *interleaving*, is defined via the concept of *independence* of events from subnets. An event e is said to be *independent* of the k-th subnet of the net under consideration, or independent of level k, if $s_k = s'_k$ for all $s = (s_K, s_{K-1}, \ldots, s_1) \in \mathcal{S}$ and $s' = (s'_K, s'_{K-1}, \ldots, s'_1) \in \mathcal{N}(e, s)$, i.e., if $\mathcal{N}_k(e, \cdot)$ is the identity function. Otherwise, e depends on the k-th subnet, or on level k. If an event depends only on a single level k, it is called a *local event* for level k; otherwise, it is a *synchronizing event* [19]. We let $First(e)$ and $Last(e)$ denote the maximum and minimum levels on which e depends. Hence, e is independent of every level k satisfying $K \geq k > First(e)$ or $Last(e) > k \geq 1$, while e might or might not depend on any level k strictly between $First(e)$ and $Last(e)$. For asynchronous systems in particular, the range of affected levels, $First(e) - Last(e) + 1$, is usually significantly smaller than K for most events e. We assume that all local events for level k are merged into a single *macro event* l_k satisfying $\mathcal{N}_k(l_k, s) =_{\text{df}} \bigcup_{e \in \mathcal{E}: First(e) = Last(e) = k} \mathcal{N}_k(e, s)$ for all $s \in \mathcal{S}$. This convention does not only simplify notation, but also improves the efficiency of our state-space generation algorithm.

Our aim is to define MDD manipulation algorithms that exploit the concept of event locality. Since an event e affects local states stored between levels $First(e)$ and $Last(e)$, firing e only causes updates of MDD nodes between these levels, plus possibly at levels higher than $First(e)$, but only when a node at level $First(e)$ becomes redundant or non-unique, and possibly levels lower than $Last(e)$, but only until recursive *Union* calls stop creating new nodes. To benefit from this observation, we need to be able to access MDD nodes by "jumping in the middle" of an MDD, namely to level $First(e)$, rather than always having to start manipulating MDDs at the root, as is done in traditional approaches, including [19]. This is the reason why

we partition the unique table, which stores MDDs, into a K-dimensional array of lists of nodes. However, two problems need to be addressed when one wants to access an MDD directly at some level $First(e)$. We treat them separately in the following two sections.

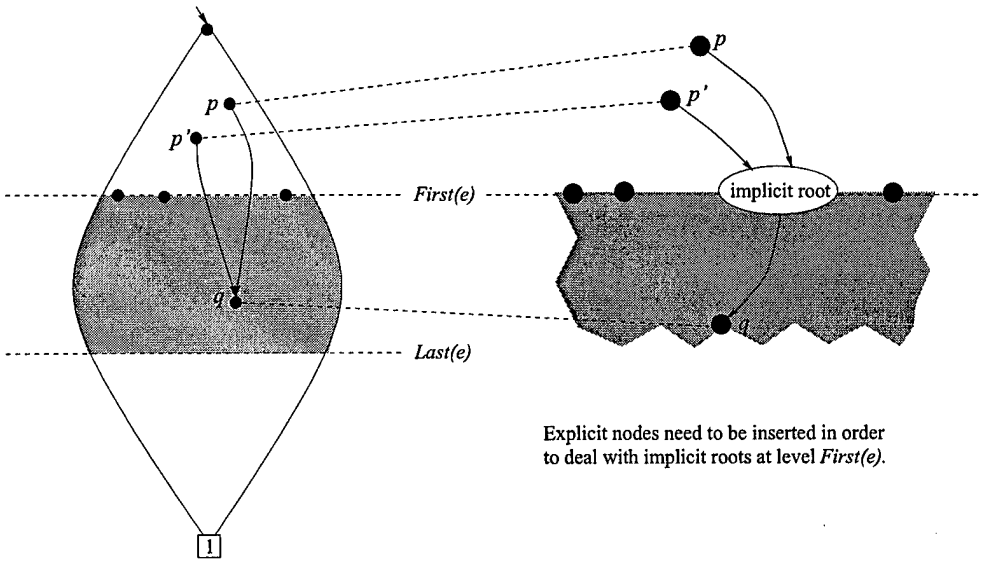

FIG. 3.1. *Illustration of event locality and the problem of implicit roots*

3.1. Implicit Roots. When one wants to explore an MDD from level $First(e)$, all nodes at this level should intuitively play the role of root nodes. However, some of them might not be represented explicitly, since redundant nodes are not stored. This happens whenever there is a node p at a level higher than $First(e)$ pointing to a node q at a level k satisfying $First(e) > k \geq Last(e)$. This situation is illustrated in Fig. 3.1, left-hand side. Conceptually, we have to re-insert these "implicit roots" at level $First(e)$ when we explore and modify the MDD due to the firing of event e. There are two approaches for doing this. The first approach stores a bag (multiset) of *upstream arcs* in each node q, corresponding to the *downstream arcs* pointing to q. In other words, for each i such that $p{\rightarrow}dw[i] = q$, there is an occurrence of p in the bag of q's upstream arcs. Implicit roots can then be detected by scanning each node stored in the unique tables for levels $First(e)+1$ through $Last(e)$, and checking whether the node possesses one or more upstream arcs to a node at a level above $First(e)$. If so, an implicit root, i.e., a redundant node, is inserted at level $First(e)$. Note that at most one implicit root needs to be inserted per node, regardless of how many arcs reach it; in our example, the arcs from both p and p' are re-routed to the same new implicit root. These redundant nodes will be deleted after firing event e, if they are still redundant. Thus, the first approach preserves the reducedness property of MDDs. Our second approach, keeps all unique redundant nodes, so that downstream arcs in the resulting MDD exist only between subsequent levels. Then, the nodes at level $First(e)$ are exactly all the nodes from which we need to start exploring the underlying MDD when firing event e. Please note that this slight variation of MDDs still possesses the fundamental property of being a *canonical* representation.

We refer to the two variants of our algorithm as *upstream-arcs approach* and *forwarding-arcs approach*; the choice for the phrase "forwarding-arcs" will become clear in the next section. The latter approach, when compared to the former, eliminates the expensive need to search for implicit roots. However, both approaches have some memory penalty potentially associated with them, the former for the storage of the

upstream arcs, which can in the worst case double the space requirements, and the latter because of the preservation of redundant nodes. We have implemented both approaches, and experimental results show that these memory overheads are compensated by a smaller peak number of MDD nodes, when compared to the approach in [19].

3.2. In-place Updates. Once all nodes at level $First(e)$, explicit as well as implicit, are detected, one can update the MDD to reflect that the firing of event e may lead to new, reachable states. Our routine *Fire* implementing this update is described in detail in Sec. 5.1. It heavily relies on the *Union* operation, as presented in Table 2.1, i.e., new MDD nodes are created and appropriately inserted, as needed. However, there is one important difference with respect to existing approaches. Our *Fire* operation stops creating new MDD-nodes as soon as it reaches level $First(e)$ when backtracking from recursive calls. At this level our algorithm just links the new sub-MDDs at the appropriate positions in the original MDD, in accordance with the concept of event locality. The only difficulty with the in-place update of some node p arises when it becomes redundant or non-unique. In the former case, p must be deleted and its incoming arcs be re-directed to its unique child node q. In the latter case, p must be deleted and its incoming arcs be re-directed to the replica node q. In the upstream-arcs approach, this is trivial since p knows its parents.

In the forwarding-arcs approach, we keep redundant nodes; thus, we eliminate p only if it becomes non-unique. However, we do not have upstream arcs. Instead of scanning all the nodes in level $First(e) + 1$ to search for arcs to p, which is a costly operation, we mark p as deleted and set a forwarding arc from p to q. The next time a node accesses p, it will update its own pointer to p, so that it points to q instead. Since node q itself might be marked as deleted later on, forwarding chains of nodes can arise. In our implementation, the nodes in these chains are deleted only after all the events at level $First(e)$ have been fired and before nodes at the next higher level are explored.

It is important to note that, although these *in-place updates* change the meaning of MDD-nodes at higher levels, they do not jeopardize the correctness of our algorithm. This is due to the interleaving semantics of asynchronous systems (cf. Sec. 5.3). Rather than performing *in-place updates*, existing approaches reported in the literature create an MDD encoding the set of global states reachable from the current states in the state space by firing event e. This is a K-level MDD, i.e., it is expensive to build compared to our sub-MDD, especially when MDDs are tall and the effect of e is restricted to a small range of levels.

Summarizing, it is the notion of event locality for asynchronous systems that allows us to drastically improve on the time efficiency of MDD-based state-space generation techniques. Exploiting locality, we can jump in and out of the "middle" of MDDs, thereby exploring only those levels that are affected by the event under investigation. While the approach reported in [19] also claims to exploit locality, it only considers some simplifications and improvements of MDD manipulations in the case of local events. However, it does not support *localized modifications* of MDDs – neither for synchronizing events, nor for local events.

4. Improving Cache Management and Iteration Control. The concept of event locality also paves the road towards significant improvements in *cache management* and *iteration control*, which we present next. An efficient cache management as well as an efficient organization of the iteration control are of utmost importance for the performance of MDD-based algorithms for state-space generation.

4.1. Intelligent Cache Management. The technique of in-place updates introduced in Sec. 3.2 allows us to enhance the efficiency of the union cache. In related work regarding state-space generation using decision

diagrams, including [19], the lifetime of the contents of the union cache cannot span more than one iteration, since the root of any MDD is deleted and re-created whenever additional reachable states are incorporated in the MDD. In other words, any change in an MDD node, i.e., in its dw-array of pointers, is really implemented as a deletion followed by an insertion.

In contrast, in our approach the "wave" of changes towards the root, caused by firing an event e, is stopped at level $First(e)$, where only a pointer is updated. This permits some union cache entries to be reused over several iterations, until the referred nodes are either changed or deleted. For this purpose, MDD nodes in our implementation have two status bits attached, namely a *cached* flag and a *dirty* flag. Instead of thoroughly cleaning up the union cache after each iteration, we can now perform a *selective purging* according to the above flags. More precisely, if an MDD node associated with a union cache entry is not deleted and if the copies present in the cache are not stale, the result may be kept in the union cache and reused later on. Experimental studies show us that the rate of reusability of union cache entries averages about 10% and that the overall performance of our algorithm can be improved by up to 13% when employing this idea.

Additionally, we devise a second optimization technique for the union cache, which is based on *prediction* and is conceptually very similar to *associative caches* studied in the field of computer architecture. Our prediction relies on the fact that if $Union(p,q)$ returns r, then also $Union(p,r)$ and $Union(q,r)$ will return r. Thus, these two additional results can be memorized in the cache, immediately after storing the entry for $Union(p,q)$. Experiments indicate that this heuristics speeds-up our algorithm by up to 12%. The reason for such a significant improvement is the following. Assume we are exploring the firing of event e in node p at level k, and assume $j \in \mathcal{N}_k(e,i)$. Then, the set of states encoded by the MDD rooted at $p \rightarrow dw[i]$ needs to be added to the set of states encoded by the MDD rooted at $p \rightarrow dw[j]$. Let r be the result of $Union(p \rightarrow dw[i], p \rightarrow dw[j])$, which becomes the new value of $p \rightarrow dw[j]$. At the next iteration, and assuming that p has not been deleted, we explore event e in node p again and, consequently, find out that e is enabled in local state i. Hence, we need to perform the update $p \rightarrow dw[j] \Leftarrow Union(p \rightarrow dw[i], p \rightarrow dw[j])$ again. However, if p has not changed, $Union(p \rightarrow dw[i], p \rightarrow dw[j])$ is identical to $Union(p \rightarrow dw[i], r) = r$. By having cached r at the previous iteration, we can avoid computing this union, even if it was never explicitly computed before.

4.2. Advanced Iteration Control. Event locality also allows us to reduce the number of iterations needed for generating reachable state spaces. Existing MDD-based algorithms for Petri nets [19, 22] fire events in some arbitrary order within each iteration, as indicated in Line 7 of function *MDDgeneration* in Table 2.2. In our version of *MDDgeneration*, however, we presort events according to function $First(\cdot)$. Our algorithm then starts at level 1 and searches for the states that can be reached from the initial state by firing all events e satisfying $First(e) = 1$ and $Last(e) \geq 1$, i.e., the macro event l_1. When reaching level k, our algorithm finds all states that can be added to the current state space by firing all events e satisfying $First(e) = k$ and $Last(e) \geq 1$, i.e., the local macro event l_k at level k and all synchronizing events that affect only level k and any level below. Moreover, in our implementation, we repeatedly fire each event at level k, as long as it is enabled and as long as firing it adds new states.

This specific sequence of firing events is essential for the correctness and efficiency of the implementation of our cache management. By working from the bottom levels to the top levels, we can clear the union and firing caches more selectively, thus, extending the lifetime of cache entries. Moreover, the access pattern to the caches is more regular and, thereby, contributes to higher hit ratios. Our firing sequence also enables delayed node deletion which allows for efficient collection and removal of non-unique and disconnected nodes, especially in the forwarding-arcs approach.

In [19], repeatedly firing events is only applied for local events, which are relatively inexpensive to process, while synchronizing events are still fired only once and in no particular order. We stress that while the new iteration control means that our iterations are potentially more expensive than those in [19], they are also potentially fewer. More precisely, our algorithm generates state spaces in at most as many iterations as the maximum *synchronizing distance* of any reachable state s, which is defined in [19] as the minimal number of synchronizing events required to reach s from the initial state, without counting local events.

5. Details of the New Algorithm. In this section, we present some important details on both variants of our new MDD-based algorithm. We first illustrate how to update MDDs in response to firing an event. We then discuss the data structures used and, finally, argue why the algorithm is correct. Please note that the complete pseudo code of the algorithm is included in the first three sections of the appendix.

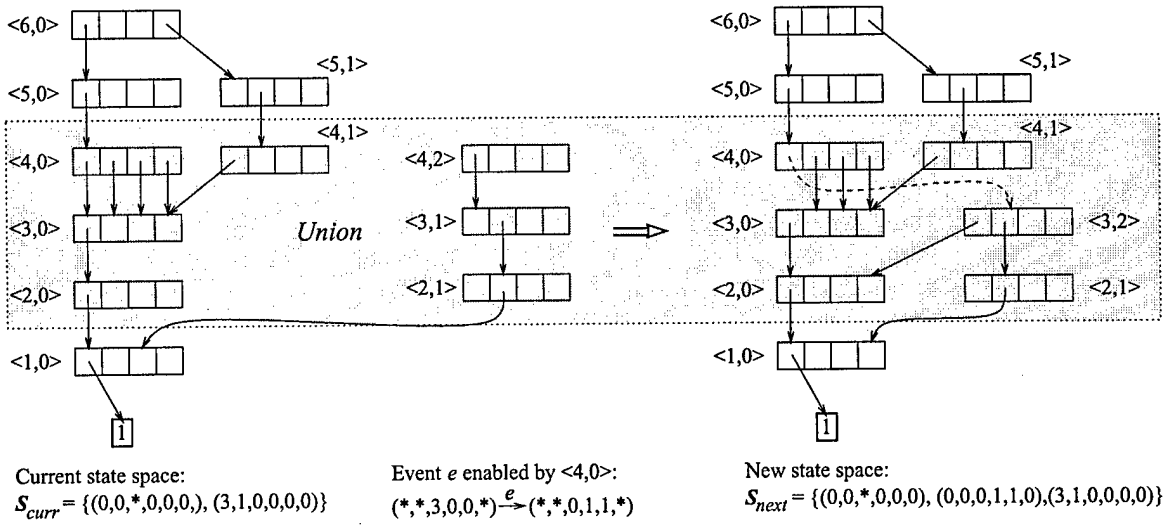

FIG. 5.1. *Example of an MDD-modification in response to firing an event*

5.1. Illustration of MDD-based Firing of Events. At each iteration of our algorithm, enabled events are fired to discover additional reachable states, which are then added to the MDD representing the currently-known portion of the reachability set of the Petri net under study. Function $Fire(e, \cdot, \cdot)$ implements this behavior with respect to event e. Fig. 5.1 illustrates, by means of a small example, how *Fire* works. The example net is partitioned into six subnets, each of them having four possible local states, numbered from 0 to 3. Hence, our MDD has six levels, and each MDD node has four downstream arcs; here, we do not draw node $\langle 0, 0\rangle$, nor any arc to it. Let the current state space, depicted on the left in Fig. 5.1, be $\mathcal{S}_{curr} = \{(0, 0, *, 0, 0, 0), (3, 1, 0, 0, 0, 0)\}$, where "$*$" stands for any local state. Assume further that event e is enabled in every state of the form $(*, *, 3, 0, 0, *)$ and that the new state reached when firing e is $(*, *, 0, 1, 1, *)$, i.e., $First(e) = 4$ and $Last(e) = 2$. Hence, if the net is in a global state described by local state 3 at level 4 and local state 0 at levels 3 and 2, event e can fire and the local states of the affected subnets are updated to 0, 1, and 1, respectively.

Exploiting event locality, our search for enabling sequences starts directly at level $First(e) = 4$. The sub-MDDs rooted at this level are searched to match the enabling pattern of e. At level 4, only the MDD rooted at $\langle 4, 0\rangle$ contains such a pattern, along the path $\langle 4, 0\rangle \xrightarrow{3} \langle 3, 0\rangle \xrightarrow{0} \langle 2, 0\rangle \xrightarrow{0} \langle 1, 0\rangle$. Then, our algorithm

generates a new MDD rooted at node $\langle 4,2\rangle$, representing the set of substates for levels 4 through 1 that can be reached from $\langle 4,0\rangle$ via e. This MDD is depicted in Fig. 5.1 in the middle. Note that only nodes at levels $First(e)$ through $Last(e)$ might have to be created, since those below $Last(e)$ can simply be linked to existing nodes, such as node $\langle 1,0\rangle$ in our example. Indeed, in our implementation, even node $\langle 4,2\rangle$ is actually not allocated, since we explore it one child at a time. This MDD corresponds to all states of the form $(\alpha,0,1,1,\beta)$, where α is any substate leading to node $\langle 4,0\rangle$ and where β is a substate reachable from the 0-th arc of node $\langle 2,0\rangle$. In our example, α and β can only be the substates $(0,0)$ and (0), respectively. In other words, the set of states to be added by firing e in node $\langle 4,0\rangle$ is $\mathcal{S}_{add} = \{(0,0,0,1,1,0)\}$. Finally, the 0-th downstream arc of node $\langle 4,0\rangle$ is updated to point to the result of the union of the MDDs rooted at nodes $\langle 3,0\rangle$ and $\langle 3,1\rangle$, which is stored in an MDD rooted at the new node $\langle 3,2\rangle$, as depicted on the right in Fig. 5.1. Hence, the resulting state space $\mathcal{S}_{next}$ is $\{(0,0,*,0,0,0), (0,0,0,1,1,0), (3,1,0,0,0,0)\}$, as desired. Note that our version of $Fire(e)$ is much more efficient than the one in [19]. In particular, it exploits the locality of e and, therefore, operates on smaller MDDs. This is important since the complexity of the *Union* operation is proportional to the number of nodes in its operand MDDs.

5.2. Implementation Details. MDD nodes store not only the addresses of their children, but also Boolean flags for garbage collection and intelligent cache management, as well as information specific to the upstream-arcs approach and to the forwarding-arcs approach.

In our implementation, nodes are stored using one *heap array* per MDD level. The pages of the heap array are created only upon request and accommodate dynamic deletion and creation of nodes. Therefore, existing nodes may not be stored contiguously in memory. For fast retrieval, we maintain a doubly-linked list of nodes. Upon deletion, a node is moved to the back of the list, thereby, allowing for garbage collection (but not garbage removal) in constant time.

The unique table, the union cache, and the firing cache are organized as arrays of hash tables, i.e., one hash table per level. For the unique table, the hash key of a node is determined using the values in its *dw*-array. For the union cache, the addresses of the two MDD nodes involved in the union are used to determine the hash key. Together with the Boolean *cached* and *dirty* flags, this allows us to reuse union cache entries across iterations without danger of accessing stale values. Finally, the hash key for firing cache entries is determined using only the address of the MDD node to which the firing operation is applied. Note that the identity of the event is implicit, since the firing cache is cleared when moving from one event to the next. The alternative approach, i.e., allowing the co-existence of entries referring to different events in the cache, would require a larger cache with a key based on a pair of MDD node and event. However, this would not bring enough benefits, since the major cost of processing the firing of an event lies in the *Union* operations, and these can indeed be cached across operations.

For the upstream-arcs approach, MDD nodes include the addresses of their parents, which we store in a bag. Our implementation uses a dynamic data structure for bags, rather than a static data structure, since the number of parents of a node is not known in advance and may be very large, in the range of several thousand nodes. While this memory overhead is still acceptable, the approach also puts a burden on time efficiency, since each update of a downstream arc must be reflected by an update of the corresponding upstream arc. Moreover, the bag of some node q only stores the address of parents p, as well as the number of indexes i such that $p{\rightarrow}dw[i] = q$, but not the indexes themselves. Thus, a linear search in the array $p{\rightarrow}dw$ must be performed to find these indexes. The alternative, namely storing these indexes in q, would require even more memory overhead.

Regarding the forwarding-arcs approach, time efficiency is improved by allowing redundant nodes to be represented explicitly. As a consequence, MDD nodes do not need to store bags of parents' addresses, but simply a counter indicating the number of incoming arcs [19]. When this counter reaches zero, it indicates that the node has become disconnected and can be deleted. Experiments show that the memory overhead of this approach, due to the storage of redundant nodes and the delayed deletion of non-unique nodes, is about the same as the memory overhead of the upstream-arcs approach. However, the forwarding-arcs approach is more time-efficient, as confirmed by the results in Sec. 6.

5.3. Correctness of the Algorithm. First of all, it is easy to see that our algorithm terminates for finite-state systems, since each iteration adds new states to the reachability set under construction. The partial correctness of our algorithm is based on the *interleaving semantics* of asynchronous systems, which formally states the following.

> Let $\mathcal{S}$ be the set of global reachable states for the system under consideration, and let $s = (s_K, s_{K-1}, \ldots, s_1) \in \mathcal{S}$ be arbitrary. Moreover, let e be an event enabled in s and $s' = (s'_K, s'_{K-1}, \ldots, s'_1)$ the global state reached by firing it. By the principle of event locality we know that $s_k = s'_k$ for all k satisfying $K \geq k > \mathit{First}(e)$ or $\mathit{Last}(e) > k \geq 1$. Then we may conclude $\hat{s} =_{\mathrm{df}} (r_K, \ldots, r_{\mathit{First}(e)+1}, s'_{\mathit{First}(e)}, \ldots, s'_{\mathit{Last}(e)}, r_{\mathit{Last}(e)-1}, \ldots, r_1) \in \mathcal{S}$, for all global states $r = (r_K, r_{K-1}, \ldots, r_1) \in \mathcal{S}$.

This interleaving principle is directly implemented in our algorithm in form of local MDD explorations and in-place updates of MDD nodes. In fact, the global state $\hat{s}$ mentioned above is implicitly inserted in our MDD whenever state s' is. There is no need to compute $\hat{s}$ explicitly, as is done in related explicit and symbolic approaches to state-space generation. This observation is the key for improving on the performance of traditional state-space generators.

6. Experimental Studies. In this section, we present several performance results regarding the two variants of our algorithm and compare them with the approach most closely related to ours, namely the one reported in [19]. The variants of our algorithm are implemented in the Petri net tool SMART (Simulation and Markovian Analyzer for Reliability and Timing) [5]. We apply the tool to the four Petri net models also considered in [19], i.e., the *dining philosophers*, the *slotted-ring* system, the *flexible manufacturing system* (FMS), and the *Kanban system*. The former two models, originally taken from [22], are composed of N identical *safe* subnets, i.e., each place contains at most one token at a time. The latter two models, originally taken from [6], have a fixed number of places and transitions, but are parameterized by the number N of initial tokens in certain places. The Petri nets for these systems are depicted in Fig. 6.1. To use MDDs, we adopt the "best" partitions found in [19]: we consider two philosophers per level and one subnet per level for the slotted-ring protocol, while we split the FMS and the Kanban system into 19 subnets (each place in a separate subnet except for $\{P_1M_1, M_1\}$, $\{P_{12}M_3, M_3\}$, and $\{P_2M_2, M_2\}$) and 4 subnets ($\{p_{mX}, p_{backX}, p_{outX}, p_X\}$ for $X = 1, 2, 3, 4$), respectively.

Table 6.1 presents several results for the two variants of our new algorithm, as well as the best-known existing algorithm [19], obtained when running SMART on a 500 MHz Intel Pentium II workstation with 512 MB of memory and 512 KB cache. For each model and choice of N, we give the size of the state space and the final number of MDD nodes, which is of course independent of the algorithm used. Then, for each algorithm, we give the peak number of MDD nodes allocated during execution, the number of iterations, and the CPU time. The peak number of MDD nodes and the number of iterations for the upstream-arcs and

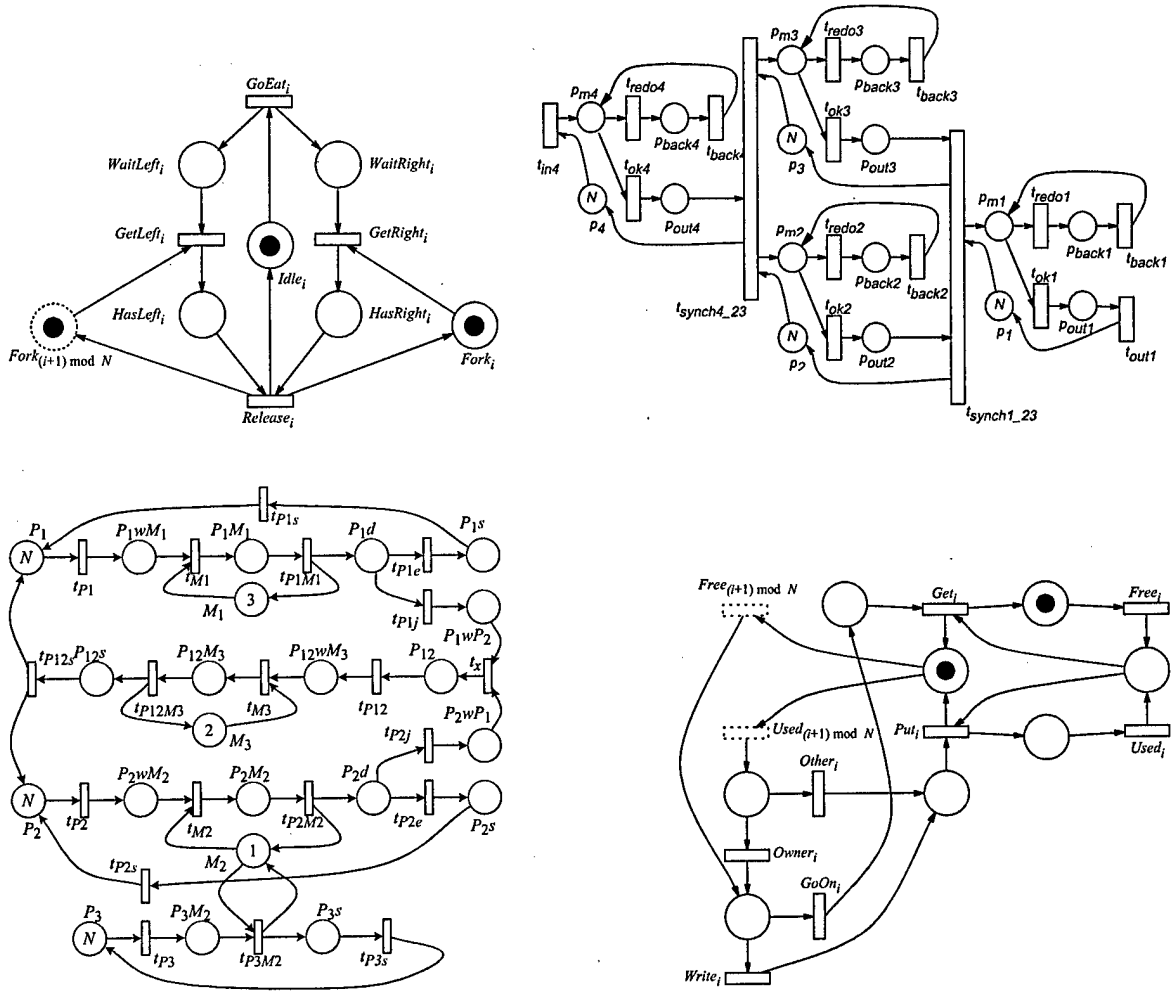

FIG. 6.1. *Petri nets used in our experiments: dining philosophers (upper left), Kanban system (upper right), FMS (lower left), and slotted-ring (lower right)*

forwarding-arcs approaches coincide, except for the FMS and the Kanban system, where the peak number reported should be increased by one for the forwarding-arcs approach. This implies that, even without introducing redundant nodes, essentially all arcs already connect nodes between adjacent levels. Thus, in our examples, the only memory overhead in the forwarding arcs approach is due to postponed node deletion.

For the models we ran, our new approach is up to one order of magnitude faster, and with few exceptions uses fewer MDD nodes than the one in [19]. The improvement mainly arises from the structural changes made to the core routine *Fire*, which reflects the notion of event locality inherent in asynchronous systems. Other improvements – most importantly our cache optimizations – contribute in average about 7–13%, and up to 22% in total, to the overall improvement in time efficiency. A comparison between the run-times for the new algorithm and the ones for the algorithm in [19] indicates an increase factor in speed ranging from approximately constant for the Kanban and FMS nets, to what appears to be almost linear (in N) for the slotted-ring model and the dining philosophers. Moreover, the forwarding-arcs approach is slightly faster than the upstream-arcs approach, except for the Kanban system on which we comment below. Since both

TABLE 6.1
Performance results

				Approach in [19]			Our new approach			
	N	$\|\mathcal{S}\|$	final nodes	peak nodes	# it.	time (sec.)	peak nodes	# it.	time (sec.) upstr.	time (sec.) fwd.
Philosophers	10	1.86×10^{6}	17	45	2	0.03	28	2	0.02	0.02
	50	2.23×10^{31}	37	285	2	0.82	168	2	0.15	0.13
	100	4.97×10^{62}	197	585	2	3.32	343	2	0.37	0.36
	200	2.47×10^{125}	397	1,185	2	13.76	693	2	1.22	1.20
	300	1.23×10^{188}	597	1,785	2	30.88	1,043	2	2.80	2.77
	400	6.10×10^{250}	797	2,385	2	60.25	1,393	2	4.52	4.40
	500	3.03×10^{313}	997	2,985	2	92.17	1,743	2	7.14	6.89
	600	1.51×10^{376}	1,197	3,585	2	121.94	2,093	2	9.33	8.93
	700	7.48×10^{438}	1,397	4,185	2	181.12	2,443	2	12.65	12.30
	800	3.72×10^{501}	1,597	4,785	2	245.76	2,793	2	16.88	16.06
	900	1.85×10^{564}	1,797	5,385	2	302.63	3,143	2	21.17	20.29
	1,000	9.18×10^{626}	1,997	5,985	2	382.04	3,493	2	26.10	24.94
Slotted ring	10	8.29×10^{9}	60	691	7	1.47	409	7	0.82	0.77
	20	2.73×10^{20}	220	4,546	12	33.32	2,328	12	12.74	12.22
	30	1.04×10^{31}	480	15,101	17	242.36	10,433	17	76.45	75.00
	40	4.16×10^{41}	840	37,066	22	1,073.64	25,374	22	297.07	293.15
	50	1.72×10^{52}	1,300	76,308	27	4,228.88	47,806	27	908.40	897.97
FMS	5	2.90×10^{6}	149	433	10	0.57	239	10	0.26	0.22
	10	2.50×10^{9}	354	1,038	15	2.42	599	15	1.05	0.88
	15	2.17×10^{11}	634	1,868	20	6.27	1,109	20	2.83	2.20
	20	6.03×10^{12}	989	2,923	25	13.52	1,769	25	6.47	4.83
	25	8.54×10^{13}	1,419	4,203	30	26.49	2,579	30	13.01	9.12
	50	4.24×10^{17}	4,694	13,978	55	209.96	8,879	55	166.28	73.13
	75	6.98×10^{19}	9,844	29,378	80	980.20	18,929	80	484.93	299.34
	100	2.70×10^{21}	16,869	50,403	105	2,681.80	32,729	105	1,448.16	845.91
Kanban	5	2.55×10^{6}	7	47	11	0.08	55	4	0.05	0.05
	10	1.01×10^{9}	12	87	21	1.26	155	4	0.66	0.76
	15	4.70×10^{10}	17	127	31	6.97	305	4	3.90	4.43
	20	8.05×10^{11}	22	167	41	24.64	505	4	15.11	16.76
	25	7.68×10^{12}	27	207	51	68.71	755	4	44.62	49.12
	30	4.99×10^{13}	32	247	61	161.49	1,055	4	113.99	123.67
	40	9.94×10^{14}	42	327	81	628.11	1,805	4	511.44	564.13
	50	1.04×10^{16}	52	407	101	1,681.96	2,755	4	1,586.32	1,492.21

variants of our new algorithm require significantly fewer peak MDD nodes, where the Kanban system is again an exception, our memory penalty is almost compensated.

The two models whose parameter N affects the height of the MDD, namely the dining philosophers and the slotted-ring model, provide a good testbed for our ideas since they give rise to tall MDDs with a high degree of event locality. For these models, the CPU times are up to 15 times faster than the ones for [19], and, more importantly, the gap widens as we continue to scale-up the nets. The main reason for this is that the number of explored nodes per event fired is much more contained in our approach, compared to [19].

When MDD heights are small, such as for the FMS and the Kanban system, our algorithm is still faster than the one in [19], but the difference is not as impressive due to our increased book-keeping overhead.

TABLE 6.2
Timing results for the Kanban net with 16 levels (one place per level)

N	1	2	3	4	5	8	10	15	20
Approach in [19] (sec.)	0.77	2.42	6.45	14.02	25.07	111.80	233.93	1,021.53	3,324.78
Upstream-arcs approach (sec.)	0.20	0.46	0.69	1.16	1.80	5.32	9.54	29.50	73.49
Forwarding-arcs approach (sec.)	0.16	0.37	0.73	1.25	1.94	5.71	10.18	29.96	69.83

The results for the Kanban system are poor compared to the ones for our other examples, although the number of iterations is reduced from $2 \cdot N + 1$ to 4 due to our advanced iteration control. There are several reasons for this. First, splitting the Kanban net into only four subnets leads to an MDD with a small depth, but a very large breadth. Clearly, any attempt to exploit locality in this case cannot have much pay-off. Second, our garbage-collection policy in the forwarding-arcs approach contributes to the proliferation of deleted nodes, which are not truly destroyed until the end of the iteration. Combined with the reduced number of iterations in our approach, the garbage collection bin grows too rapidly. Usually, late node deletion is beneficial, since doing garbage collection in bulks reduces the number of times nodes are scanned for removal. However, in case of the Kanban system, we see how this can backfire. It is worth noting that using a finer and not particularly "good" partition of the Kanban net, with one place per level, drastically changes the results, as shown in Table 6.2. We only need to scale-up the model to $N = 20$ to see an improvement of about factor 50 with respect to [19]. This observation indicates that our algorithm might be well-suited in cases when a good partitioning cannot be found automatically or by hand, e.g., due to insufficient heuristics.

Summarizing, our algorithm performs much better than [19] when Petri nets are partitioned into many subnets, thereby leading to tall MDDs, as the exploitation of event locality becomes more beneficial. The memory overhead in our approach, which is due to larger-sized MDD nodes in case of the upstream-arcs approach and to redundant and deleted, but not-yet-destroyed nodes, in case of the forwarding-arcs approach, is almost accounted for in practice by the small peak number of MDD nodes.

7. Related Work. A variety of approaches for the generation of reachable state spaces of synchronous and asynchronous systems have been suggested in the literature, where state spaces are represented either in an *explicit* or in a *symbolic* way.

Explicit state-space generation techniques build the reachable state space of the system under consideration by successively iterating its next-state function [3, 6, 10, 13]. To achieve space efficiency, various techniques have been introduced. Two techniques, namely *multi-level data structures* and *merging common bitvectors*, deserve special mentioning. Multi-level data structures exploit the structure of the underlying representation of the system under consideration, e.g., the approach reported in [6] and implemented in [5] is based on a decomposition of a Petri net into subnets. As the name suggests, merging common bitvectors aims at compressing the storage needed for each state – a bitvector – by merging common sub-bitvectors [3]; indeed, the result is somewhat analogous to the one obtained using BDDs. The latter technique is also successfully used in automata-based model-checking tools [13]. While explicit methods still require space linear in the number of states, they usually possess advantages for numerical state-space analyses, e.g., those based on Kronecker algebra [7], which may directly work on data structures employed for explicit state storage.

To avoid the problem of state-space explosion when building the explicit state space of concurrent, asynchronous systems, researchers have developed three key techniques. (i) *Compositional minimization techniques* build the state space of a concurrent system stepwise, i.e., parallel component by parallel component, and minimize the state space of each intermediate system according to a behavioral congruence or an interface specification [12]. (ii) *Partial-order techniques* exploit the fact that several traces of an asynchronous system may be equivalent with respect to the properties of interest [11]; thus, it is sufficient to explore only a single trace of each equivalence class. (iii) *Techniques exploiting symmetries in systems* – such as those with repeated sub-systems – can be used to avoid the explicit construction of symmetric subgraphs of the overall state spaces [9]; colored Petri nets are also an example of this aspect [14].

Symbolic state-space generation techniques have traditionally focused on (synchronous) hardware systems rather than on (asynchronous) software systems [1, 4, 15, 17]. In the Petri net community, they were first applied by Pastor et al. in [22]. This paper developed a BDD-based algorithm for the generation of the reachability sets of safe Petri nets, by encoding each place of a net as a Boolean variable. The algorithm is capable of generating state spaces of very large Petri nets within hours [24]. In recent work, Pastor and Cortadella introduced a more efficient encoding of Petri nets by exploiting place invariants [21]. However, the underlying logic is still based on Boolean variables. In contrast, our work uses a more general version of decision diagrams, namely MDDs [15, 19], by which the amount of information carried in a single node of a decision diagram can be increased. In particular, MDDs allow for a straightforward encoding of arbitrary, i.e., not necessarily safe, Petri nets. Since we have already compared our approach to related MDD-based techniques in the previous sections, we refrain from a repetition of this comparison here.

8. Conclusions and Future Work. This paper presented a very efficient new algorithm for building the reachable state spaces of asynchronous systems. As in previous work [19], state spaces are symbolically represented via Multi-valued Decision Diagrams (MDDs), which – unlike Binary Decision Diagrams – are able to store complex information within a single node. However, in contrast to previous work, our algorithm fully exploits event locality in asynchronous systems, integrates an intelligent cache management, and achieves faster convergence via an advanced iteration control. Analytical results of examples well-known in the Petri net community show that our algorithm is often about one order of magnitude faster than the one introduced in [19] – which in turn improves on previous algorithms – with only a relatively small decrease in space efficiency. In summary, our approach successfully reduces the run-time penalty of related algorithms when generating very large state spaces using symbolic storage techniques. To the best of our knowledge, our algorithm is the first symbolic one taking advantage of event locality.

Regarding future work, we intend to parallelize our algorithm for shared-memory and distributed-memory architectures. The idea is to map different levels of MDDs to different processors and, thereby, to speed-up the state-space construction further while being able to store larger MDDs on distributed architectures. Our algorithm is particularly suited for this kind of parallelization since all data structures are already split according to levels. We believe that our approach promises to avoid the run-time penalties for parallelization reported in the literature [18, 23, 25], especially regarding distributed-memory implementations on networks of workstations and PC clusters.

Acknowledgments. We would like to thank A.S. Miner for many fruitful discussions on multi-valued decision diagrams and on implementations of state-space generation techniques, as well as César Muñoz for carefully proofreading a draft of this paper.

REFERENCES

[1] R. BRYANT, *Graph-based algorithms for Boolean function manipulation*, IEEE Transactions on Computers, 35 (1986), pp. 677–691.

[2] ——, *Symbolic Boolean manipulation with ordered binary-decision diagrams*, ACM Computing Surveys, 24 (1992), pp. 393–418.

[3] P. BUCHHOLZ, *Hierarchical structuring of superposed GSPNs*, in 7th International Workshop on Petri Nets and Performance Models (PNPM '97), St. Malo, France, June 1997, IEEE Computer Society Press, pp. 81–90.

[4] J. BURCH, E. CLARKE, K. MCMILLAN, D. DILL, AND L. HWANG, *Symbolic model checking:* 10^{20} *states and beyond*, Information and Computation, 98 (1992), pp. 142–170.

[5] G. CIARDO AND A. MINER, *SMART: Simulation and Markovian Analyzer for Reliability and Timing*, in IEEE International Computer Performance and Dependability Symposium (IPDS '96), Urbana-Champaign, IL, USA, September 1996, IEEE Computer Society Press, p. 60.

[6] ——, *Storage alternatives for large structured state spaces*, in 9th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (Tools '97), R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, eds., vol. 1245 of Lecture Notes in Computer Science, St. Malo, France, June 1997, Springer-Verlag, pp. 44–57.

[7] G. CIARDO AND M. TILGNER, *On the use of Kronecker operators for the solution of generalized stochastic Petri nets*, Tech. Report 96-35, Institute for Computer Applications in Science and Engineering, Hampton, VA, USA, May 1996.

[8] E. CLARKE, E. EMERSON, AND A. SISTLA, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Programming Languages and Systems, 8 (1986), pp. 244–263.

[9] E. CLARKE, T. FILKORN, AND S. JHA, *Exploiting symmetry in model checking*, in Computer Aided Verification (CAV '93), C. Courcoubetis, ed., vol. 697 of Lecture Notes in Computer Science, Elounda, Greece, June/July 1993, Springer-Verlag, pp. 450–462.

[10] D. DILL, *The Murphi verification system*, in Computer Aided Verification (CAV '96), R. Alur and T. Henzinger, eds., vol. 1102 of Lecture Notes in Computer Science, New Brunswick, NJ, USA, July 1996, Springer-Verlag, pp. 390–393.

[11] P. GODEFROID, *Partial-order Methods for the Verification of Concurrent Systems – An Approach to the State-explosion Problem*, vol. 1032 of Lecture Notes in Computer Science, Springer-Verlag, 1996.

[12] S. GRAF, B. STEFFEN, AND G. LÜTTGEN, *Compositional minimisation of finite state systems using interface specifications*, Formal Aspects of Computing, 8 (1996), pp. 607–616.

[13] G. HOLZMANN, *The model checker Spin*, IEEE Transactions on Software Engineering, 23 (1997), pp. 279–295. Special issue on Formal Methods in Software Practice.

[14] K. JENSEN, *Coloured Petri nets*, in Petri Nets: Central Models and Their Properties, Part I, Proceedings of an Advanced Course, vol. 254 of Lecture Notes in Computer Science, Bad Honnef, Germany, September 1987, Springer-Verlag, pp. 248–299.

[15] T. KAM, T. VILLA, R. BRAYTON, AND A. SANGIOVANNI-VINCENTELLI, *Multi-valued decision diagrams: Theory and applications*, Multiple-Valued Logic, 4 (1998), pp. 9–62.

[16] P. KEMPER, *Numerical analysis of superposed GSPNs*, IEEE Transactions on Software Engineering, 22 (1996), pp. 615–628.

[17] K. McMillan, *Symbolic Model Checking: An Approach to the State-explosion Problem*, PhD thesis, Carnegie-Mellon University, 1992.

[18] K. Milvang-Jensen and A. Hu, *BDDNOW: A parallel BDD package*, in Second International Conference on Computer-Aided Design (FMCAD '98), G. Gopalakrishnan and P. Windley, eds., vol. 1522 of Lecture Notes in Computer Science, Palo Alto, CA, USA, November 1998, pp. 501–512.

[19] A. Miner and G. Ciardo, *Efficient reachability set generation and storage using decision diagrams*, in 20th International Conference on Application and Theory of Petri Nets (ICATPN '99), J. Kleijn and S. Donatelli, eds., vol. 1639 of Lecture Notes in Computer Science, Williamsburg, VA, USA, June 1999, Springer-Verlag, pp. 6–25.

[20] T. Murata, *Petri nets: Properties, analysis and applications*, Proceedings of the IEEE, 77 (1989), pp. 541–579.

[21] E. Pastor and J. Cortadella, *Efficient encoding schemes for symbolic analysis of Petri nets*, in IEEE Conference on Design, Automation and Test in Europe (DATE '98), Paris, France, March 1998, IEEE Computer Society Press, pp. 790–795.

[22] E. Pastor, O. Roig, J. Cortadella, and R. Badia, *Petri net analysis using Boolean manipulation*, in 15th International Conference on the Application and Theory of Petri Nets (ICATPN '94), R. Valette, ed., vol. 815 of Lecture Notes in Computer Science, Springer-Verlag, June 1994, pp. 416–435.

[23] R. Ranjan, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vincentelli, *Binary decision diagrams on network of workstations*, in IEEE International Conference on Computer Design (ICCD '96), Austin, TX, USA, October 1996, IEEE Computer Society Press, pp. 358–364.

[24] O. Roig, J. Cortadella, and E. Pastor, *Verification of asynchronous circuits by BDD-based model checking of Petri nets*, in 16th International Conference on the Application and Theory of Petri Nets (ICATPN '95), G. De Michelis and M. Diaz, eds., vol. 815 of Lecture Notes in Computer Science, Springer-Verlag, June 1995, pp. 374–391.

[25] T. Stornetta and F. Brewer, *Implemention of an efficient parallel BDD package*, in 33rd Design Automation Conference (DAC '96), Las Vegas, NV, USA, June 1996, Association for Computing Machinery, pp. 641–644.

Appendix A. Data Types and General Purpose Routines.

This section contains the definitions of the data types used in our pseudo code, as well as some general-purpose routines for operating on these data types.

A.1. Data types. The data types employed by our algorithm are the following:

- The type *level* of levels is $[1..K]$, where K is a positive integer constant encoding the number of MDD levels. The constant N of type array$[1..K]$ of *int* represents the branching degree of MDDs for each level.
- The type *event* for events is integer, i.e., event names are encoded as integer values. Functions $First(e : event) : level$ and $Last(e : event) : level$ return the index of the first and last level affected by the corresponding event, respectively. Procedure *PreprocessEvents*() sorts the events according to their first affected level, in increasing order. In the pseudo code, we also use the notation $e' < e$ to indicate that e' is smaller than e according to the order imposed by *PreprocessEvents*. Note that all local events for level k are merged into a single macro event l_k.
- The type $mddNode(k : level)$ for MDD nodes is a record type having the following fields:
  - dw : array $[0..(N[k]-1)]$ of *mddAddr*, which stores the $N[k]$ downstream arcs for all K children of the node under consideration.
  - up : bag of *mddAddr*, which stores upstream arcs.
  - *cached* : *boolean*, a flag indicating whether there exists a cache entry referring to this node.
  - *dirty* : *boolean*, a flag signaling in combination with *cached* whether the cached copies are stale.
- The *mddAddr* type is a "virtual" address of an MDD node, which is a pair $\langle lvl, ind \rangle$ represented by a 32-bit integer. The first $\lceil \log_2 K \rceil$ bits of an address encode the level *lvl* of a node, the remaining bits encode the position *ind* of the node within that level. In the pseudo code, $\mathcal{S}(p : mddAddr)$ denotes the state space represented by the MDD rooted at p.
- The storage for "physical" MDD nodes is a vector $\mathcal{T}[1..K]$ of heap-arrays, one heap-array per level. Upon request, memory for $\mathcal{T}$ is allocated dynamically by pages. We use a 1024 node page size. In the pseudo code, the memory allocation and release procedures are denoted by $AllocateMemory(k : level) : mddAddr$ and $ReleaseMemory(p : mddAddr)$, respectively. These nodes are also accessible through a linked list which allows separate fast access to the deleted and the non-deleted nodes.
- The unique table (UT), $UT[1..K]$, is an array of hash tables, one hash table per level, which store pointers to unique MDD nodes. The hash key of an MDD node is computed solely over the values of the dw-pointers.
- The union cache (UC), $\mathcal{U}[1..k]$, and firing cache (FC), $\mathcal{F}[1..K]$, are hash tables that store the results of already computed operations. A UC table entry has type $(\{p : mddAddr, q : mddAddr\}, r : mddAddr)$, while a FC table entry has type $(p : mddAddr, r : mddAddr)$. Also, for the union cache, if $k = \max(p.lvl, q.lvl)$, then the triplet is hashed in $UC[k]$.

For all hash tables mentioned above, the size of a table is dynamic, i.e., insertions and deletions may re-dimension it. If the number of elements reaches the size of the table, we enlarge the table to about twice its current size (more precisely, to the next prime number larger than twice the current size). If the number of elements is less than $\frac{1}{4}th$ of the size of the table, we shrink the table to about half its current size (more precisely, to the next prime number smaller than half the current size). The table size is not increased if it is larger than a preset upper bound, and is not decreased if it is smaller than a preset lower bound.

A.2. Routines for Managing the Unique Table.

- *InsertInUT*(in p : *mddAddr*, out r : *mddAddr*) : *boolean*
  Searches $UT[p.lvl]$ for a node with the same pattern of downstream arcs as $p{\rightarrow}dw$. If it is found, r is set to the address of this node, and the function returns *true*. Otherwise, p is added to $UT[p.lvl]$, r is left unchanged, and the function returns *false*.

- *RemoveFromUT*(in p : *mddAddr*)
  Removes p from $UT[p.lvl]$.

- *ClearUT*(in k : *level*)
  Clears all entries in $UT[k]$.

A.3. Routines for Managing the Union Cache.

- *LookUpInUC*(in k : *level*, in p : *mddAddr*, in q : *mddAddr*, out r : *mddAddr*)
  Searches $\mathcal{U}[k]$ for an element of the form $(\{p,q\},\cdot)$. If such a $(\{p,q\},x)$ is found, it sets r to x and returns *true*. Otherwise, it leaves r unchanged and returns *false*. The result is independent of the order in which the two parameters p and q are supplied.

- *InsertInUC*(in k : *level*, in p : *mddAddr*, in q : *mddAddr*, in r : *mddAddr*)
  Inserts $(\{p,q\},r)$ in $\mathcal{U}[k]$. Given the logic of our algorithm, $\mathcal{U}[k]$ does not contain any element of the form $(\{p,q\},\cdot)$ and $k = \max\{p.lvl, q.lvl\}$. The effect on $\mathcal{U}[k]$ is independent of the order in which the two parameters p and q are supplied.

- *RemoveFromUC*(in k : *level*, in p : *mddAddr*, in q : *mddAddr*)
  Removes the entry of the form $(\{p,q\},\cdot)$ from $\mathcal{U}[k]$.

- *ClearUC*(in k : *level*)
  Clears all entries in $\mathcal{U}[k]$.

A.4. Routines for Managing the Firing Cache.

- *LookUpInFC*(in k : *level*, in p : *mddAddr*, out r : *mddAddr*) : *boolean*
  Searches $\mathcal{F}[k]$ for an element of the form $(p,\cdot)$. If such a (p,x) is found, it sets r to x and returns *true*. Otherwise, it leaves r unchanged and returns *false*.

- *InsertInFC*(in k : *level*, in p : *mddAddr*, in r : *mddAddr*)
  Inserts (p,r) in $\mathcal{F}[k]$. Given the logic of our algorithm, $\mathcal{F}[k]$ does not contain any element of the form $(p,\cdot)$ and $k \geq p.lvl$.

- *ClearFC*(in k : *level*)
  Clears all entries in $\mathcal{F}[k]$.

A.5. Routines for Managing Sets and Bags. Sets of integers are implemented as *queues*. Elements can be picked from the head (FIFO) and from the tail (LIFO) of a queue, according to the desired strategy.

- *PickAnyElement*(inout $\mathcal{L}$: set of *int*) : *int*
  Selects and removes an arbitrary element from set $\mathcal{L}$, and returns it.

Bags of *mddAddr* are implemented as linked lists of pairs (*mddAddr*, *count*). They are managed via the following two functions:

- *AddElement*(in p : *mddAddr*, in *plus* : *int*, inout b : *bag*)
  If bag b contains p, the count of p is increased by *plus*. Otherwise, p is added with count *plus* to the list of elements in b.

- *RemoveElement*(in p : *mddAddr*, in *minus* : *int*, inout b : *bag*)
  If bag b contains p with count greater than or equal to *minus*, the routine subtracts *minus* from the count of p, followed by the deletion of p in case the count becomes 0. Otherwise, b is left unchanged.

Often the symbols "$\in$", "$=$", "$\neq$", and "$\emptyset$" are also used in the context of bags, with their obvious meanings.

A.6. Routines for handling events.

- *NewStates*(in k : *level*, in e : *event*, in i : *int*) : *set* of *int*
  Returns the set of local states obtained by firing event e, when e is enabled by local state i at level k. Basically, this routine is the local next-state function for the subnet encoded in level k.

- *IsIndependent*(in k : *level*, in e : *event*) : *boolean*
  An event e is independent of some level k when the firing of e leaves level k unchanged. Note that this property is different from e being disabled.

A.7. Modifications for the Forwarding-arcs Approach. In the forwarding-arcs approach, the record field *up* of node type *mddNode* is replaced by the following two fields:

- *in* of type *int*, which stores the number of incoming arcs from the next higher level, and
- *deleted* of type *boolean*, which signals whether the node has been marked for deletion. If so, it is redundant, and its unique forwarding arc is stored in $dw[0]$.

Moreover, since downstream arcs do not skip levels, entries of $\mathcal{U}[k]$ and $\mathcal{F}[k]$ refer only to nodes at level k. Thus, the level parameter k can be removed for routines *LookUpInUC*, *InsertInUC*, *RemoveFromUC*, *LookUpInFC*, and *InsertInFC*.

Appendix B. Detailed Pseudo Code for the "Upstream-arcs" Variant.

MDDgeneration(in m : array[1..K] of *int*) : *mddAddr*

Generates the state space of a model with respect to the initial state m and returns the address of the MDD's root.

```
local k : level;
local e : event;
local q : mddAddr;
local mddChanged : boolean;                    • flag signaling whether more iterations are needed

 1. for k = 1 to K do
 2.     ClearUT(k);                             • the UT is cleared only once at the beginning
 3. for k = 1 to K − 1 do
 4.     ClearUC(k);                             • the UC is initialized here and later purged of out-of-date entries
 5.     ClearFC(k);                             • the FC is cleared here and at the end of each Fire
 6. q ⇐ SetInitial(m);
 7. PreprocessEvents();                         • sort events in increasing order regarding First(·)
 8. repeat
 9.     mddChanged ⇐ false;                     • true if any node of the MDD changes in this iteration
10.     for k = 1 to K do
11.         foreach event e satisfying First(e) = k do
12.             Fire(e, q, mddChanged);
13. until mddChanged = false;
14. return q;
```

Fire(in e : *event*, in s : *mddAddr*, inout *mddChanged* : *boolean*)

Generates and inserts the states reachable from the currently known state space represented by s via event e. For any node at level $First(e)$, it calls *FireFromFirst*, which propagates work downstream by calling *Union* and *FireRecursive*. Then, for any node at a level k, with $First(e) > k \geq Last(e)$, having incoming downstream arcs from a level above $First(e)$, *Fire* creates a temporary redundant node at level $First(e)$, and calls *FireFromFirst* on it. The dummy node is removed at the end, if, after exploration, it is still redundant. This second phase must be performed after the first one, to avoid re-exploring (formerly redundant) nodes just introduced. The flag *mddChanged* is passed through and updated.

```
local k : level;
local i : int;
local p, q, r, d : mddAddr;
local pHasDummy : boolean;                     • signals if an implicit root has to be inserted

 1. foreach p ∈ T[First(e)] do                 • fire e starting at nodes in level First(e)
 2.     if FireFromFirst(e, p) then
 3.         mddChanged ⇐ true;
 4. for k = Last(e) to First(e) − 1 do         • check for downstream arcs skipping over First(e)
 5.     foreach p ∈ T[k] do
 6.         pHasDummy ⇐ false;
 7.         foreach q ∈ p→up do
 8.             if q.lvl > First(e) then       • downstream arc from q to p skips over First(e)
 9.                 if not pHasDummy then
10.                     d ⇐ CreateNode(First(e), p);   • insert a redundant node d at level First(e) pointing to p
11.                     InsertInUT(d, null);
12.                     pHasDummy ⇐ true;      • d is not in the UT, since it is a redundant node
13.                 for i = 0 to N[q.lvl] do   • find all downstream arcs from q to p and re-direct them to d
14.                     if q→dw[i] = p then
15.                         SetArc(q, i, d);
```

(to be continued on next page)

```
                                                              (continued from previous page)
16.        if pHasDummy then                          • if a redundant node has been created, explore it
17.          if not FireFromFirst(e, d) then          • if it is unchanged, it is still redundant...
18.            RemoveFromUT(d);                       • ...remove d from the UT and...
19.            CheckNode(d);                          • ...re-direct to p any arc that was re-directed to d, then delete d
20. for k = First(e) − 1 downto 1 do                  • must clean up in this order for this to work
21.    foreach ({p, q}, r) ∈ U[k] do
22.      if (p→up = ∅) or (q→up = ∅) or (r.lvl > 0 and r→up = ∅)      • disconnected nodes...
23.      or (p→dirty) or (q→dirty) or (r.lvl > 0 and r→dirty) then    • ...and out-of-date entries...
24.        RemoveFromUC(k, p, q);                     • ...are removed from the UC
25.    foreach p ∈ T[k] do                            • clear disconnected nodes at level k
26.      DeleteDownstream(p);
27. for k = Last(e) to First(e) − 1 do                • clear firing caches at levels below First(e)
28.    ClearFC(k);
```

FireFromFirst(in e : *event*, in p : *mddAddr*) : *boolean*

Fires event e starting from node p in the UT, satisfying $p.lvl = First(e)$. It propagates work downstream by calling *Union* and *FireRecursive*. It returns *true*, if node p was changed, and *false*, otherwise. If p changes, its address is removed from the UT. Moreover, either the node itself is deleted, if it has become redundant, or p is re-inserted in the UT (this allows for updating its hash value). If the node is removed, the change is propagated upstream using *CheckNode*. If the node is not changed, p is left in the UT.

```
local L : set of int;
local pHasChanged : boolean;                          • flag signaling whether MDD with root p has changed
local f, u : mddAddr;
local i, j : int;

 1. L ⇐ LocalStatesToExplore(p, e);                   • get all the local states that potentially enable e
 2. pHasChanged ⇐ false;
 3. while L ≠ ∅ do
 4.    i ⇐ PickAnyElement(L);                         • choose any element i in L and remove it from L
 5.    f ⇐ FireRecursive(First(e) − 1, e, p→dw[i]);   • this call returns p→dw[i] if e is local
 6.    if f ≠ ⟨0, 0⟩ do                               • f = ⟨0, 0⟩ if and only if e could not fire
 7.       foreach j ∈ NewStates(First(e), e, i) do    • j is a local state reachable from i when firing e
 8.          u ⇐ Union(f, p→dw[j]);
 9.          if u ≠ p→dw[j] then                     • the firing of e added new states
10.             if not pHasChanged then              • this is the first change to p in this call
11.                RemoveFromUT(p);                  • p must be removed from the UT before changing it
12.                pHasChanged ⇐ true;               • remember not to remove p from the UT again
13.             if NewStates(First(e), e, j) ≠ ∅ then
14.                AddElement(j, L);                 • j needs to be explored (possibly again)
15.             SetArc(p, j, u);
16. if pHasChanged then
17.    if p→cached then p→dirty ⇐ true;              • cache entries referring to p are stale
18.    CheckNode(p);                                 • put back p into the UT, or delete it
19. return pHasChanged;
```

FireRecursive(in k : *level*, in e : *event*, in p : *mddAddr*) : *mddAddr*

Returns the address of a node representing the set of states reachable from $\mathcal{S}(p)$ when event e occurs, ignoring the dependency of e on levels above $p.lvl$. Function *FireRecursive* propagates work only downstream, since it only changes a temporary node t in-place. The returned value is guaranteed to be in the UT, unless it is not $\langle 0,1\rangle$ or $\langle 0,0\rangle$.

```
local  L : set of int;
local  r, t, f : mddAddr;
local  atSameLevel : boolean;                          • p.lvl = k ?
local  i, j : int;

 1. if k < Last(e) then                                • the end of the recursion is reached
 2.     return p;
 3. if p.lvl < k and IsIndependent(k, e) then          • e does not depend on level k
 4.     return FireRecursive(k − 1, e, p);             • continue at the next level
 5. if LookUpInFC(k, p, r) then
 6.     return r;
 7. t ⇐ CreateNode(k, ⟨0, 0⟩);                         • create a temporary node t
 8. if p.lvl < k then                                  • at this point, e depends on k
 9.     atSameLevel ⇐ false;
10.     L ⇐ LocalStatesEnablingEvent(k, e);            • initialize the set L to all local states enabling e
11. else                                               • k = p.lvl
12.     atSameLevel ⇐ true;
13.     L ⇐ LocalStatesToExplore(p, e);                • initialize the set L to all reachable local states enabling e
14. while L ≠ ∅ do
15.     i ⇐ PickAnyElement(L);                         • choose any element i in L and remove it from L
16.     if atSameLevel then                            • find states reachable from p→dw[i] via e
17.         f ⇐ FireRecursive(k − 1, e, p→dw[i]);
18.     else                                           • nothing to explore here; move on to the next level
19.         f ⇐ FireRecursive(k − 1, e, p);
20.     if f ≠ ⟨0, 0⟩ then                             • f = ⟨0,0⟩ if and only if e could not fire
21.         foreach j ∈ NewStates(k, e, i) do
22.             u ⇐ Union(f, t→dw[j]);
23.             if u ≠ t→dw[j] then                    • the firing of e in p→dw[i] added new states
24.                 if NewStates(k, e, j) ≠ ∅ then     • e is still enabled
25.                     AddElement(j, L);              • j will have to be explored (possibly again)
26.                 SetArc(t, j, u);
27. t ⇐ CheckNode(t);                                  • since t→up = ∅, this cannot cause recursive deletes upstream
28. InsertInFC(k, p, t);
29. return t;
```

Union(in p : *mddAddr*, in q : *mddAddr*) : *mddAddr*

Returns the address r of the node representing $\mathcal{S}(p) \cup \mathcal{S}(q)$. It uses and updates the UC to speed-up computation. The returned value is guaranteed to be in the UT, unless it is not $\langle 0,1\rangle$ or $\langle 0,0\rangle$. Of course, $r.lvl \leq Max(p.lvl, q.lvl)$.

```
local  k : level;
local  i : int;
local  r, u : mddAddr;

 1. if p = ⟨0, 1⟩ or q = ⟨0, 1⟩ return ⟨0, 1⟩;        • deal with special cases first
 2. if p = ⟨0, 0⟩ or p = q return q;
 3. if q = ⟨0, 0⟩ return p;
```

(to be continued on next page)

(continued from previous page)

```
4.  k ⇐ Max(p.lvl, q.lvl);
5.  if LookUpInUC(k, p, q, r) then                  • if found, result of the union is returned in r
6.      return r;
7.  r ⇐ CreateNode(k, ⟨0, 0⟩);                      • otherwise, the union is computed in r
8.  for i = 0 to N[k] − 1 do
9.      if k > p.lvl then                           • p is at a lower level than q
10.         u ⇐ Union(p, q→dw[i]);
11.     else if k > q.lvl then                      • q is at a lower level than p
12.         u ⇐ Union(p→dw[i], q);
13.     else                                        • p and q are at the same level
14.         u ⇐ Union(p→dw[i], q→dw[i]);
15.     SetArc(r, i, u);
16. r ⇐ CheckNode(r);          • since r→up = ∅, this cannot cause recursive deletes upstream
17. InsertInUC(k, p, q, r);                         • record the result of this union in the UC
18. if p ≠ r then InsertInUC(k, p, r, r);           • add predicted cache requests
19. if q ≠ r then InsertInUC(k, q, r, r);
20. p→cached, q→cached, r→cached ⇐ true;
21. return r;
```

CheckNode(in p : *mddAddr*) : *mddAddr*

Enforces the MDD properties for node p, which is not in the UT. It ensures that this node is neither redundant nor a replica. If so, p is inserted in the UT. Otherwise, node p is disconnected from upstream nodes and deleted by calling *DeleteUpstream*, which in turn calls *CheckNode* on these nodes, and so on. The recursion stops when a modified node does not have to be deleted. Function *CheckNode* returns the address of the node representing the set of states initially described by p, and this address is guaranteed to be in the UT. As we allow a redundant root node, we treat it as a special case.

```
local x : mddAddr;
1.  if p.lvl = K then                                   • check special case
2.      InsertInUT(p, null);                            • put the root node back into the UT
3.      return p;                  • this allows for keeping the root node even if it is redundant
4.  if p→dw[0] = p→dw[1] = ··· = p→dw[N[k] − 1] then    • p is redundant; delete it and use its child
5.      x ⇐ p→dw[0];
6.      DeleteUpstream(p, x);                           • all downstream arcs pointing to p must now point to x
7.      return x;
8.  else if InsertInUT(p, x) then                       • p is a replica of x; delete it and use x instead
9.      DeleteUpstream(p, x);                           • all downstream arcs pointing to p must now point to x
10.     return x;
11. else
12.     return p;                                       • p is a distinct node and was inserted in the UT
```

DeleteUpstream(in o : *mddAddr*, in n : *mddAddr*)

Changes any downstream arc pointing to the old node o, not present in the UT, so that it points to the new node n instead, thus disconnecting node o. Then it deletes o. After changing the downstream arcs of any node p in the upstream bag of o, it removes p from the UT and enforces the reducedness property on it by calling *CheckNode*, which in turn may call *DeleteUpstream*.

```
local p : mddAddr;
local i : int;
```

(to be continued on next page)

(continued from previous page)

```
1. foreach p ∈ o→up do                          • check all nodes directly upstream
2.     RemoveFromUT(p);                         • updated node will be deleted or re-inserted in the UT by CheckNode
3.     for i = 0 to N[p.lvl] − 1 do
4.         if p→dw[i] = o then
5.             SetArc(p, i, n);                 • this call does not need a downstream recursion
6.     CheckNode(p);                            • enforce reducedness property
7. for i = 0 to N[o.lvl] do
8.     SetArc(o, i, ⟨0, 0⟩);                    • disconnect node...
9. ReleaseMemory(o);                            • ...and kill it
```

SetArc(in p : *mddAddr*, in i : *int*, in n : *mddAddr*)

Sets the i-th downstream arc of node p to n, while maintaining consistency with the upstream arcs.

```
local o : mddAddr;
1. o ⇐ p→dw[i];                                 • old node pointed by the downstream arc
2. p→dw[i] ⇐ n;                                 • re-direct downstream arc
3. if n.lvl ≠ 0 then                            • no need to link ⟨0, 0⟩ or ⟨0, 1⟩
4.     AddElement(p, 1, n→up);                  • increase count of upstream arcs for the new node pointed to
5. if o.lvl ≠ 0 then                            • no need to unlink ⟨0, 0⟩ or ⟨0, 1⟩
6.     RemoveElement(p, 1, o→up);               • reduce count of upstream arcs for old node
```

DeleteDownstream(in p : *mddAddr*)

If node p has no incoming arcs, this routine removes p from the UT and deletes it, after having recursively examined each of its downstream arcs.

```
local q : mddAddr;
local i : int;
1. if p→up = ∅ then
2.     RemoveFromUT(p);
3.     for i = 0 to N[p.lvl] do
4.         q ⇐ p→dw[i];
5.         SetArc(p, i, ⟨0, 0⟩);                • disconnect old downstream arc pointing to q
6.         DeleteDownstream(q);                 • check if q still has incoming arcs
7.     ReleaseMemory(p);                        • kill node p
```

SetInitial(in m : array $[1..K]$ of *int*) : *mddAddr*

Constructs the MDD representing the initial state m of the model, and returns a pointer to the MDD's root.

```
local p, q : mddAddr;
local k : int;
1. q ⇐ ⟨0, 1⟩;                                  • initialize q to node ⟨0, 1⟩
2. for k = 1 to K do
3.     p ⇐ CreateNode(k, ⟨0, 0⟩);
4.     SetArc(p, m[k], q);                      • link new node, at level k, to the one below, at level k − 1
5.     InsertInUT(p, null);                     • use null because p is known to be a new node
6.     q ⇐ p;
7. return q;
```

CreateNode(in k : *level*, in *initial* : *mddAddr*) : *mddAddr*

Allocates a level-k node with all the entries in dw initialized to *initial*, up initialized to $\emptyset$, flags *cached* and *dirty* initialized to *false*, and returns its address. It also updates the bag of upstream arcs for node *initial*.

```
local p : mddAddr;
local i : int;

1. p ⇐ AllocateMemory(k);
2. p→up ⇐ ∅;
3. for i = 0 to N[k] − 1 do
4.     p→dw[i] ⇐ initial;
5. if initial.lvl > 0 then
6.     AddElement(p, N[k], initial→up);
7. p→cached, p→dirty ⇐ false;
8. return p;
```

LocalStatesEnablingEvent(in k : *level*, in e : *event*) : *set* of *int*

Returns the set of local states at level k which enable e.

```
local L : set of int;
local i : int;

1. L ⇐ ∅;
2. for i = 0 to N[k] − 1 do
3.     if NewStates(k, e, i) ≠ ∅ then
4.         AddElement(i, L);
5. return L;
```

- refer to the local next-state function of the underlying model

LocalStatesToExplore(in p : *mddAddr*, in e : *event*) : set of *int*

Returns the set of local states at level $p.lvl$ which (1) are currently reachable via the considered path from the root to p and (2) enable e. If e is independent of level $p.lvl$, only Condition (1) is restrictive, since $NewStates(p.lvl, e, i) = \{i\}$, i.e., all local states at this level enable e.

```
local L : set of int;
local i : int;

1. L ⇐ ∅;
2. for i = 0 to N[p.lvl] − 1 do
3.     if p→dw[i] ≠ ⟨0, 0⟩ and NewStates(p.lvl, e, i) ≠ ∅ then
4.         AddElement(i, L);
5. return L;
```

Appendix C. Detailed Pseudo Code for the "Forwarding-arcs" Variant.

Routines *SetInitial*, *LocalStatesEnablingEvent*, and *LocalStatesToExplore* are as for the upstream-arcs approach. The other routines are given here, including some new ones.

MDDgeneration(in m : array[1..K] of *int*) : *mddAddr*

Generates the state space of a model with respect to initial state m, and returns the address of the MDD's root.

```
local k : level;
local q : mddAddr;
local mddChanged : boolean;                    • flag signaling whether more iterations are needed
 1. for k = 1 to K do
 2.     ClearUT(k);                             • the UT is cleared only once at the beginning
 3. for k = 1 to K − 1 do
 4.     ClearUC(k);                             • the UC is initialized here and later purged of out-of-date entries
 5. for k = 1 to K − 1 do
 6.     ClearFC(k);                             • the FC is cleared here and at the end of each Fire
 7. q ⇐ SetInitial(m);
 8. PreprocessEvents();                         • sort events in increasing order regarding First(·)
 9. repeat
10.     mddChanged ⇐ false;                    • true if any node changes in this iteration
11.     Fire(l_1);                             • fire the local macro event at level 1
12.     for k = 2 to K do
13.         DeleteForwarding(k);               • eliminate non-unique nodes at level k
14.         foreach event e satisfying First(e) = k do
15.             Fire(e, q, mddChanged);
16. until mddChanged = false;
```

DeleteForwarding(in k : *level*)

Removes all nodes marked for deletion at level $k-1$ and destroys the corresponding forwarding chain, after appropriately re-directing the downstream arcs from nodes at level k. This requires to remove these nodes from the UT and to check them back in. Thus, this procedure might cause nodes at level k to become marked for deletion.

```
local p, u : mddAddr;
local pHasChanged : boolean;                   • flag signaling whether p has changed
local i : int;
 1. foreach p ∈ T[k] do                        • eliminate forwarding arcs and nodes marked for deletion at level k − 1
 2.     pHasChanged ⇐ false;
 3.     for i = 0 to N[k] − 1 do
 4.         if p→dw[i].lvl > 0 then
 5.             u ⇐ UpdateArc(p, i);           • update arc p→dw[i], in case it points to a node marked for deletion
 6.             if u ≠ p→dw[i] then            • p→dw[i] does point to a node marked for deletion
 7.                 if pHasChanged = false then
 8.                     RemoveFromUT(p);       • p must be removed from the UT before changing it
 9.                     pHasChanged ⇐ true;    • remember not to remove p from the UT again
10.                 SetArc(p, i, u);           • point p→dw[i] to the equivalent node not marked for deletion
11.     if pHasChanged then
12.         CheckNode(p);                      • this might mark p for deletion
```

UpdateArc(in p : *mddAddr*, in i : *int*) : *mddAddr*

If $q = p{\rightarrow}dw[i]$ is not marked for deletion, q is returned. Otherwise, u is returned – where u is the "ultimate" node in the forwarding chain – after (1) either re-directing q's forwarding arcs to u, so that further accesses to q will determine u more efficiently, or (2) deleting q, if one just followed the last arc reaching it (either downstream or forwarding).

```
local q, u : mddAddr;

 1. q ⇐ p→dw[i];
 2. if (q.lvl > 0) and q→deleted then
 3.     u ⇐ UpdateArc(q, 0);                    • the only arc pointing to q is followed...
 4.     if q→in = 1 then
 5.         SetArc(p, 0, ⟨0, 0⟩);
 6.         ReleaseMemory(q);                   • ...so q can finally be deleted
 7.         SetArc(q, 0, u);      • q cannot be deleted, but its forwarding arc can be set to the end of the chain
 8.     if q→in = 0 then
 9.         return u;                           • u is not marked for deletion
10. else
11.     return q;
```

Fire(in e : *event*, in q : *mddAddr*, inout *mddChanged* : *boolean*)

Generates and inserts the states reachable from the current state space via event e. For any node at level $First(e)$, it calls *FireFromFirst*. The flag *mddChanged* is passed through and updated.

```
local k : level;
local p, q, r : mddAddr;

 1. foreach p ∈ 𝒯[First(e)] do                  • fire e starting at nodes in the first level affecting it
 2.     if FireFromFirst(e, p) then
 3.         mddChanged ⇐ true;
 4. for k = First(e) − 1 downto 1 do            • must clean up in this order for this to work
 5.     foreach ({p, q}, r) ∈ 𝒰[k] do
 6.         if (p→up = ∅) or (q→up = ∅) or (r→up = ∅)        • disconnected nodes...
 7.         or (p→dirty) or (q→dirty) or (r→dirty) then      • ...and out-of-date entries...
 8.             RemoveFromUC(k, p, q);                       • ...are removed from the UC
 9.     foreach p ∈ 𝒯[k] do                     • clear disconnected nodes at level k
10.         DeleteDownstream(p);
11. for k = Last(e) to First(e) − 1 do          • clear firing caches at levels below First(e)
12.     ℱ[k] ⇐ ∅;
```

FireFromFirst(in e : *event*, in p : *mddAddr*) : *boolean*

Fires event e starting from node p in the UT, satisfying $p.lvl = First(e)$. It propagates work downstream by calling *Union* and *FireRecursive*. It returns *true* if node p was changed, and *false*, otherwise. If the node changes, p is removed from the UT. Moreover, whether node p is deleted, if it has become redundant, or p is re-inserted in the UT (this allows for the hash value to be updated). If the node is removed, the change is recorded by *CheckNode* using a forwarding arc. If the node is not changed, p is left in the UT.

```
local ℒ : set of int;
local pHasChanged : boolean;          • flag signaling whether p has changed
local f, u : mddAddr;
local i, j : int;
```

(to be continued on next page)

(continued from previous page)

```
1.  L ⇐ LocalStatesToExplore(p, e);                    • get all the local states that potentially enable e
2.  pHasChanged ⇐ false;
3.  while L ≠ ∅ do
4.      i ⇐ PickAnyElement(L);                         • choose any element i in L and remove it from L
5.      f ⇐ FireRecursive(e, p→dw[i]);                 • this call returns p→dw[i] if e is local
6.      if f ≠ ⟨0,0⟩ then                              • f = ⟨0,0⟩ if and only if e could not fire
7.          foreach j ∈ NewStates(First(e), e, i) do   • j is a local state reachable from i when firing e
8.              u ⇐ Union(f, p→dw[j]);
9.              if u ≠ p→dw[j] then                    • the firing of e added new states
10.                 if not pHasChanged then            • this is the first change to p in this call
11.                     RemoveFromUT(p);               • we must remove p from the UT before changing it
12.                     pHasChanged ⇐ true;            • remember not to remove p from the UT again
13.                 if NewStates(First(e), e, j) ≠ ∅ then   • if e is still enabled...
14.                     AddElement(j, L);              • ...j will have to be explored (possibly again)
15.                 SetArc(p, j, u);
16. if pHasChanged then
17.     if p→cached then p→dirty ⇐ true;               • cache entries referring to p are invalidated
18.     CheckNode(p);                                  • put p back into the UT, or delete it
19. return pHasChanged;
```

FireRecursive(in e : *event*, in p : *mddAddr*) : *mddAddr*

Returns the address of a node representing the set of states reachable from $\mathcal{S}(p)$ when event e occurs, ignoring the dependency of e on levels above $p.lvl$. *FireRecursive* propagates work only downstream, since it only changes a temporary node t in-place. Because redundant nodes are preserved, the returned value is guaranteed to be in the UT and at the same level as p, unless it is $\langle 0,0 \rangle$.

```
local  L : set of int;
local  r, t, f : mddADdr;
local  i, j : int;

1.  if p.lvl < Last(e) then                            • end of the recursion
2.      return p;
3.  if LookUpInFC(p.lvl, p, r) then
4.      return r;
5.  r ⇐ CreateNode(p.lvl, ⟨0,0⟩);                      • create a temporary node t
6.  L ⇐ LocalStatesToExplore(p, e);                    • initialize the set L to all reachable local states enabling e
7.  while L ≠ ∅ do
8.      i ⇐ PickAnyElement(L);                         • choose any element i in L and remove it from L
9.      f ⇐ FireRecursive(e, p→dw[i]);                 • find states reachable from p→dw[i] via e
10.     if f ≠ ⟨0,0⟩ then                              • f = ⟨0,0⟩ if and only if e could not fire
11.         foreach j ∈ NewStates(p.lvl, e, i) do
12.             u ⇐ Union(f, r→dw[j]);
13.             if u ≠ r→dw[j] then                    • the firing of e in p→dw[i] added new states
14.                 if NewStates(p.lvl, e, j) ≠ ∅ then
15.                     AddElement(j, L);              • j will have to be explored (possibly again)
16.                 SetArc(r, j, u);
17. r ⇐ CheckNode(r);                                  • since t→up = ∅, this cannot cause recursive deletes upstream
18. InsertInFC(p.lvl, p, r);
19. return t;
```

Union(in p : *mddAddr*, in q : *mddAddr*) : *mddAddr*

Returns the address r of the node representing $\mathcal{S}(p) \cup \mathcal{S}(q)$, where $p.lvl = q.lvl$. It uses and updates the UC to speed-up computation. Since redundant nodes are kept, the returned value is guaranteed to be in the UT and at the same level of p and q, unless it is not $\langle 0, 0 \rangle$.

```
local r, u : mddAddr;
local i : int;
1.  if p = ⟨0,1⟩ or q = ⟨0,1⟩ return ⟨0,1⟩;          • deal with special cases first
2.  if p = ⟨0,0⟩ and q = ⟨0,0⟩ return ⟨0,0⟩;
3.  if p = q return q;
4.  if LookUpInUC(p.lvl, p, q, r) then              • if found, result of the union is returned in r
5.      return r;
6.  r ⇐ CreateNode(p.lvl, ⟨0,0⟩);                   • otherwise, the union is computed in r
7.  for i = 0 to N[p.lvl] − 1 do
8.      u ⇐ Union(p→dw[i], q→dw[i]);
9.      SetArc(r, i, u);
10. r ⇐ CheckNode(r);                               • since r→up = ∅, this cannot cause recursive deletes upstream
11. InsertInUC(p.lvl, p, q, r);                     • record the result of this union in the UC
12. if p ≠ r then InsertInUC(k, p, r, r);           • add predicted cache requests
13. if q ≠ r then InsertInUC(k, q, r, r);
14. p→cached, q→cached, r→cached ⇐ true;
15. return r;
```

CheckNode(in p : *mddAddr*) : *mddAddr*

Ensures that p, which is not in the UT, is not a replica or a redundant node pointing to $\langle 0, 0 \rangle$, and inserts p in the UT. Otherwise, node p is deleted (if it has no incoming arcs) or it is marked for deletion and a forwarding arc is placed in it. If it has incoming arcs, this can happen only when *CheckNode* is called from *FireFromFirst*, i.e., never when p is a dummy pointing to $\langle 0, 0 \rangle$. In any case, *CheckNode* returns the address of the node representing the set of states initially described by p. This address is guaranteed to be in the UT, unless it is not $\langle 0, 0 \rangle$.

```
local q : mddAddr;
local i : int;
1.  if p→dw[0] = p→dw[1] = ··· = p→dw[N[p.lvl] − 1] = ⟨0,0⟩ then   • this can happen only when p→in = 0
2.      ReleaseMemory(p);                           • p is a dummy with downstream arcs pointing to ⟨0,0⟩, delete it
3.      return ⟨0,0⟩;
4.  else if InsertInUT(p, q) then                   • p is redundant, remove it, and use u instead
5.      for i = 0 to N[p.lvl] do
6.          SetArc(p, i, ⟨0,0⟩);                    • disconnect old downstream arcs
7.      if p→in = 0 then                            • p can now be deleted
8.          ReleaseMemory(p);
9.      else                                        • deletion of p must be delayed
10.         p→deleted ⇐ true;                       • mark p for future deletion
11.         SetArc(p, 0, q);                        • record the forwarding arc
12.     return q;
13. else
14.     return p;                                   • p is a distinct node and was inserted in the UT
```

SetArc(in p : *mddAddr*, in i : *int*, in n : *mddAddr*)

Sets the i-th downstream arc of node p to n, while at the same time maintaining consistency with the incoming-arcs count. If the incoming-arcs count of the old node $p{\rightarrow}dw[i]$ becomes 0, this node will be removed later.

local o : *mddAddr*;

```
1. o ⇐ p→dw[i];                          • old node pointed by downstream arc
2. p→dw[i] ⇐ n;                          • re-direct downstream arc
3. if n.lvl ≠ 0 then                     • no need to keep track of ⟨0,0⟩ or ⟨0,1⟩
4.     n→in ⇐ n→in + 1;                  • increase count of upstream arcs for new node
5. if o.lvl ≠ 0 then                     • no need to keep track of ⟨0,0⟩ or ⟨0,1⟩
6.     o→in ⇐ o→in − 1;                  • reduce count of upstream arcs for old node
```

DeleteDownstream(in p : *mddAddr*)

If node p has no incoming arcs, this routine removes p from the UT and deletes it, after having recursively examined each of its downstream arcs.

local q : *mddAddr*;
local i : *int*;

```
1. if p→in = 0 then
2.     RemoveFromUT(p);
3.     for i = 0 to N[p.lvl] do
4.         q ⇐ p→dw[i];
5.         if q.lvl > 0 then
6.             SetArc(p, i, ⟨0,0⟩);          • disconnect old downstream arc pointing to q
7.             DeleteDownstream(q);          • check if q still has incoming arcs
8.     ReleaseMemory(p);
```

CreateNode(in k : *level*, in *initial* : *mddAddr*) : *mddAddr*

Allocates a level-k node with all the entries in dw initialized to *initial*, in initialized to zero, flags *cached* and *dirty* initialized to *false*, and returns its address. It also updates the incoming-arcs count for node *initial*.

local p : *mddAddr*;
local i : *int*;

```
1. p ⇐ AllocateMemory(k);
2. p→in ⇐ 0;
3. for i = 0 to N[k] − 1 do
4.     p→dw[i] ⇐ initial;
5. if initial.lvl > 0 then
6.     initial→in ⇐ initial→in + N[k];
7. p→cached, p→dirty ⇐ false;
8. return p;
```

Appendix D. An Illustration of the Algorithm.

In this section, we illustrate the upstream-arcs variant of our algorithm by means of a small example, namely the Kanban net depicted in Fig. 6.1, upper right, with $N = 1$ for the initial marking.

D.1. Partitioning and Initialization. In order to apply our MDD-based approach, the Kanban net is partitioned into four subnets, subnet 1 to subnet 4. Subnet i contains the places p_i, p_{m_i}, p_{back_i}, and p_{out_i}. The local macro event l_{i-1} consists of transitions t_{ok_i}, t_{redo_i}, and t_{back_i} – plus transition t_{out_1} or t_{in_4} where applicable (cf. Fig. 6.1). Further, the system possesses two synchronizing events, t_{synch1_23} and t_{synch4_23}, which we abbreviate by s_1 and s_2, respectively.

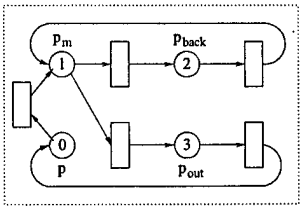

FIG. D.1. *Indexing of local states for the Kanban subnets*

Given this partitioning, each subnet has four local states which we number 0 through 3 as show in Figure D.1. In the sequel, we also abuse notation and write (I_4, I_3, I_2, I_1), where $I_j \subseteq \{0,1,2,3\}$ and $1 \leq j \leq 4$, for the set $\{(i_4, i_3, i_2, i_1) \mid i_4 \in I_4,\ i_3 \in I_3,\ i_2 \in I_2,\ i_1 \in I_1\}$. If $I_j = \{0,1,2,3\}$, we write $I_j = *$ for short.

TABLE D.1
Transitions of the example net.

macro event l_1					
*	*	*	*	*	*
*	*	*	*	*	*
*	*	*	*	*	*
1	{2,3}	2	1	3	0

macro event l_2			
*	*	*	*
*	*	*	*
1	{2,3}	2	1
*	*	*	*

macro event l_3			
*	*	*	*
1	{2,3}	2	1
*	*	*	*
*	*	*	*

macro event l_4					
0	1	1	{2,3}	2	1
*	*	*	*	*	*
*	*	*	*	*	*
*	*	*	*	*	*

syn. s_1	
*	*
3	0
3	0
0	1

syn. s_2	
3	0
0	1
0	1
*	*

The initial state (initial marking) of our net is $(0,0,0,0)$. Thus, the initially reachable states space $\mathcal{S}$ is $\{(0,0,0,0)\}$, which is represented by the MDD depicted in Fig. D.2, left-hand side. The net's transitions are schematically shown in the six tables of Table D.1, one table per event. Each column of a table contains an enabling pattern of the considered event (on the left), i.e., a set of global states, and the global state resulting after the event fires (on the right).

In the following, we show how the upstream-arcs variant of our algorithm constructs the reachable state space of the Kanban net. Before the iterative work of the algorithm starts, the routine *PreprocessEvents* sorts the events in the order $l_1 < l_2 < l_3 < s_1 < l_4 < s_2$, since $First(l_1) = 1$, $First(l_2) = 2$, $First(l_3) = First(s_1) = 3$, and $First(l_4) = First(s_2) = 4$.

D.2. First Iteration. In the first iteration, starting from the MDD representing the initial state, our algorithm invokes the following routines:

1. *Fire*(l_1), which attempts a *FireFromFirst*($l_1, \langle 1, 0\rangle$): unsuccessful, no local states enable l_1.
2. *Fire*(l_2), which attempts a *FireFromFirst*($l_2, \langle 2, 0\rangle$): unsuccessful, no local states enable l_2.
3. *Fire*(l_3), which attempts a *FireFromFirst*($l_3, \langle 3, 0\rangle$): unsuccessful, no local states enable l_3.
4. *Fire*(s_1), which attempts a *FireFromFirst*($s_1, \langle 3, 0\rangle$): unsuccessful, no local states enable s_1.

The first enabled event is local macro event l_4, as confirmed by routine *FireFromFirst*($l_4, \langle 4, 0\rangle$) which finds local state 0 enabling l_4. Since *NewStates*$(4, l_4, 0)$ is $\{1\}$, the downstream pointers of node $\langle 4, 0\rangle$ are updated to include the state $(1, 0, 0, 0)$ reached by firing l_4, i.e., $\langle 4, 0\rangle.dw[1] \Leftarrow Union(\langle 4, 0\rangle.dw[0], \langle 4, 0\rangle.dw[1]) = Union(\langle 3, 0\rangle, \langle 0, 0\rangle) = \langle 3, 0\rangle$.

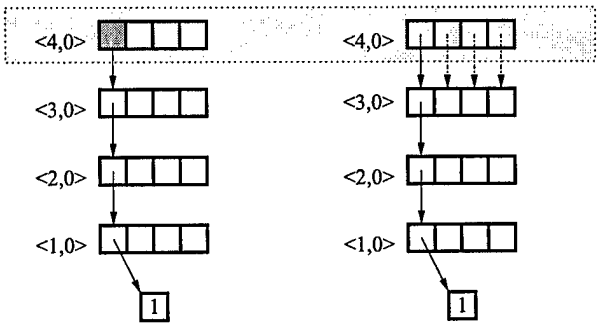

FIG. D.2. *Iteration* 1, *event* l_4

However, the firing of l_4 is not exhausted, yet, since local state 1 on level 4 still enables l_4. After repeating the above exploration scheme three times, all downstream arcs of node $\langle 4, 0\rangle$ point to node $\langle 3, 0\rangle$. Thus, the reachable state space $\mathcal{S}$ discovered so far is updated to $\mathcal{S} \cup \{(\{1, 2, 3\}, 0, 0, 0)\} = \{(*, 0, 0, 0)\}$ (cf. Fig. D.2, right-hand side).

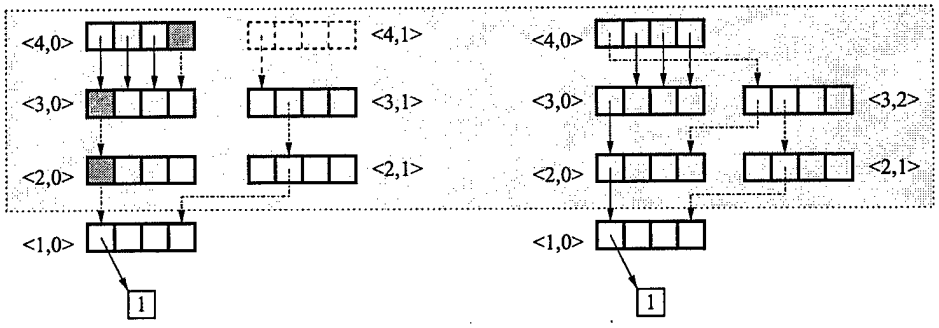

FIG. D.3. *Iteration* 1, *event* s_2

Next, the synchronizing event s_2 becomes enabled due to the sequence $\langle 4, 0\rangle \xrightarrow{3} \langle 3, 0\rangle \xrightarrow{0} \langle 2, 0\rangle \xrightarrow{0} \langle 1, 0\rangle$ (cf. Fig. D.3). The routine *FireFromFirst* called with respect to node $\langle 4, 0\rangle$ builds the MDD rooted at $\langle 3, 1\rangle$ in a series of *FireRecursive* calls:

1. *FireRecursive*$(2, s_2, \langle 2, 0\rangle) = \langle 2, 1\rangle$ and $\langle 2, 1\rangle.dw[1] = \langle 1, 0\rangle$
2. *FireRecursive*$(3, s_2, \langle 3, 0\rangle) = \langle 3, 1\rangle$ and $\langle 3, 1\rangle.dw[1] = \langle 2, 1\rangle$

In Fig. D.3, left-hand side, also a node $\langle 4, 1\rangle$ is depicted, which is not actually created. The purpose of showing it is to complete the representation of $\{(0, 1, 1, 0)\}$, which is the new state obtained by firing s_2. By calling *Union*

regarding nodes $\langle 3,0\rangle$ and $\langle 3,1\rangle = \langle 3,2\rangle$, the new state is added to $\mathcal{S}$. Hence, $\mathcal{S}$ is updated to $\mathcal{S} \cup \{(0,1,1,0)\} = \{(*,0,0,0),(0,1,1,0)\}$. Moreover, node $\langle 3,2\rangle$ is linked as the 0-th successor of $\langle 4,0\rangle$, in order to bind the new MDD to the pattern enabling the firing. Since the temporary MDD rooted at $\langle 3,1\rangle$, which is used for computing the union, is disconnected, it is removed by the next *DeleteDownstream* call, which concludes the first iteration.

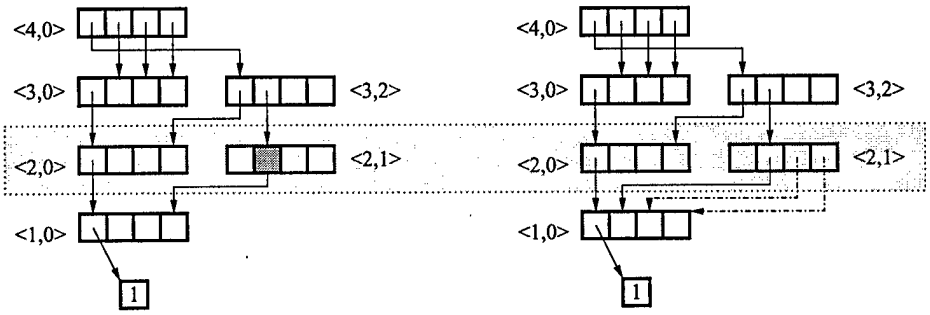

FIG. D.4. *Iteration 2, event* l_2

D.3. Second Iteration. In the second iteration, local macro event l_2 is detected to be enabled by local state 1 in node $\langle 2,1\rangle$. However, the exploration from node $\langle 2,0\rangle$ is still unsuccessful, since this node is unchanged. Event l_2 fires twice (cf. Fig. D.4, right-hand side) and adds local states 2 and 3 to node $\langle 2,1\rangle$. Hence, the updated reachable state space $\mathcal{S}$ is $\{(*,0,0,0),(0,1,\{1,2,3\},0)\}$.

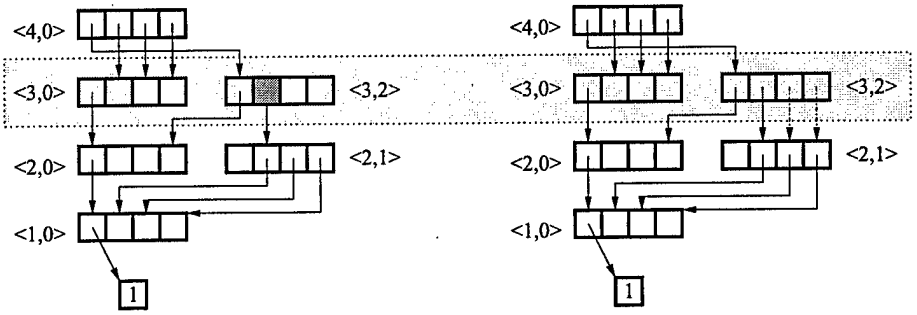

FIG. D.5. *Iteration 2, event* l_3

Similarly, local macro event l_3 is enabled by only one node at level 3, namely node $\langle 3,2\rangle$. After firing it twice from node $\langle 3,2\rangle$, the state space $\mathcal{S}$ becomes $\{(*,0,0,0),(0,\{1,2,3\},\{1,2,3\},0)\}$ (cf. Fig. D.5).

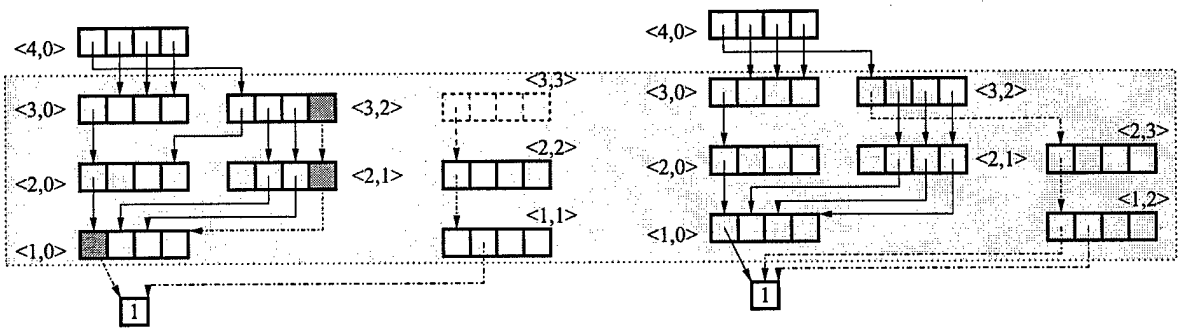

FIG. D.6. *Iteration 2, event* s_1

The new states added so far contribute to the enabling of synchronizing event s_1. Its exploration, initiated at level 3, finds the sequence $\langle 3,2\rangle \xrightarrow{3} \langle 2,1\rangle \xrightarrow{3} \langle 1,0\rangle \xrightarrow{0} \langle 0,1\rangle$. This path represents the set of states $\{(*,3,3,0)\}$. In the current state space, this pattern is part of only one global state, state $\{(0,3,3,0)\}$. By firing s_1, the new state $\{(0,0,0,1)\}$ is reached. Next, *Union* is invoked regarding the MDDs rooted at nodes $\langle 2,1\rangle$ and $\langle 2,2\rangle$. The resulting MDD, rooted at node $\langle 2,3\rangle$, is linked as the 0-th successor of node $\langle 3,2\rangle$ (cf. Fig. D.6). Thus, the new state is integrated in the state space. Hence, $\mathcal{S} = \{(*,0,0,0),(0,\{1,2,3\},\{1,2,3\},0),(0,0,0,1)\}$.

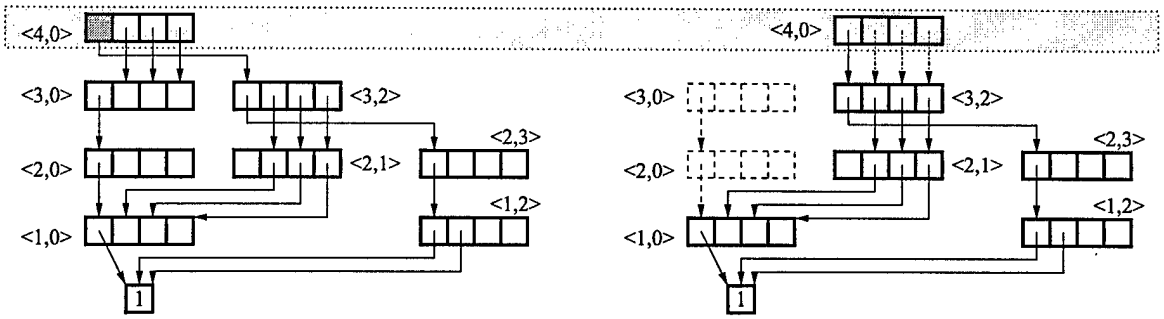

FIG. D.7. *Iteration 2, event l_4*

Exploration of local macro event l_4 from the only node at level 4, node $\langle 4,0\rangle$, reveals some new states that need to be incorporated in our MDD. To do so, all the local states of node $\langle 4,0\rangle$ have to be searched. Since there exists a transition from local state 0 to local state 1, as part of the macro event, node $FireRecursive(3,l_4,\langle 4,0\rangle.dw[0]) = \langle 3,2\rangle$ has to be added to $\langle 4,0\rangle.dw[1]$, since $NewStates(4,l_4,0) = \{1\}$. Accordingly, the call $Union(\langle 3,0\rangle,\langle 3,2\rangle)$ creates the new node $\langle 3,3\rangle$ and sets its downstream pointers to $Union(\langle 3,0\rangle.dw[i],\langle 3,2\rangle.dw[i])$, for $0 \le i \le 3$. The results are $\langle 2,3\rangle = Union(\langle 2,0\rangle,\langle 2,3\rangle)$, $\langle 2,1\rangle$, $\langle 2,1\rangle$, and $\langle 2,1\rangle$, respectively. Node $\langle 3,\rangle$ is then looked up in the unique table and identified as node $\langle 3,2\rangle$, which is already hashed. Hence, $Union(\langle 3,0\rangle,\langle 3,2\rangle)$ returns the address to node $\langle 3,2\rangle$ and stores this result in the union cache. As a consequence, $\langle 4,0\rangle.dw[1]$ is set to point to node $\langle 3,2\rangle$.

Next, local state 0 is explored, and l_4 is found to remain enabled. The following calls subsequently set all the downstream arcs of $\langle 4,0\rangle$ to $\langle 3,2\rangle$. The steps are the same as illustrated before; the only exception is that the result of $Union(\langle 3,0\rangle,\langle 3,2\rangle)$ is looked up and found in the union cache, without being computed. When all four downstream arcs of $\langle 4,0\rangle$ are updated, its old child $\langle 3,0\rangle$ becomes disconnected and has to be removed. The routine *DeleteDownstream* will do this by scanning the branch all the way down to $\langle 1,0\rangle$ (cf. Fig. D.7). At the end of $Fire(l_4)$, the discovered reachable state space is $\mathcal{S} = \{(*,0,0,\{0,1\}),(*,\{1,2,3\},\{1,2,3\},0)\}$.

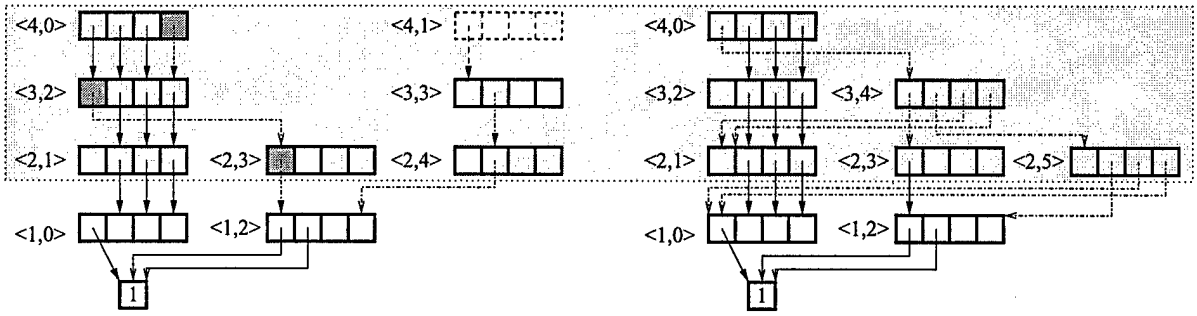

FIG. D.8. *Iteration 2, event s_2*

In the next step, event s_2 is detected to be enabled; it can fire from $\langle 4,0\rangle \xrightarrow{3} \langle 3,2\rangle \xrightarrow{0} \langle 2,3\rangle \xrightarrow{0} \langle 1,2\rangle$. Hence, $FireRecursive(3,s_2,\langle 3,2\rangle)$ builds the nodes representing the outcome of firing s_2, i.e., "$\ldots \xrightarrow{0} \langle 3,3\rangle \xrightarrow{1} \langle 2,4\rangle \xrightarrow{1} \langle 1,2\rangle$."

Then, $Union(\langle 3,2\rangle, \langle 3,3\rangle)$ adds the result to the state space. The process of creating the sub-MDD rooted at $\langle 3,4\rangle$ involves several recursive calls:

1. $\langle 3,4\rangle.dw[0] = Union(\langle 2,3\rangle, \langle 0,0\rangle) = \langle 2,3\rangle$, a hashed node
2. $\langle 3,4\rangle.dw[1] = Union(\langle 2,1\rangle, \langle 2,4\rangle) = \langle 2,5\rangle$, a new node
3. $\langle 3,4\rangle.dw[2] = Union(\langle 2,1\rangle, \langle 0,0\rangle) = \langle 2,1\rangle$, a hashed node
4. $\langle 3,4\rangle.dw[3] = Union(\langle 2,1\rangle, \langle 0,0\rangle) = \langle 2,1\rangle$, a hashed node

To complete the execution of $FireFromFirst(s_2, \langle 4,0\rangle)$, node $\langle 3,4\rangle$ is linked to the MDD as 0-th child of node $\langle 4,0\rangle$, the node where the enabling pattern originated. The state space $\mathcal{S}$ now incorporates the new states $\{0,1,1,\{0,1\}\}$, i.e., $\mathcal{S} = \{(*,0,0,\{0,1\}), (*,\{1,2,3\},\{1,2,3\},0), (0,1,1,\{0,1\})\}$. This completes the second iteration.

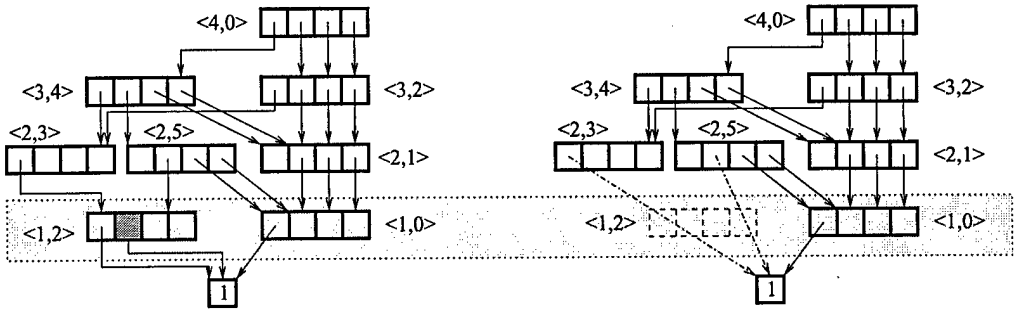

FIG. D.9. *Iteration* 3, *event* l_1

D.4. Third Iteration. As opposed to the first two iterations, event l_1 is enabled in the third iteration. The new node $\langle 1,2\rangle$ created in the previous phase has a non-zero pointer in local state 1. Event l_1 can fire twice and sets the last two downstream pointers of node $\langle 1,2\rangle$ to point to node $\langle 0,1\rangle$. When the firing is exhausted, node $\langle 1,2\rangle$ is tested for redundancy. The routine *CheckNode* finds that all the children of the node are equal and that the node is not the root, i.e., it is a redundant. To eventually preserve the reducedness property of MDDs, *CheckNode* will in turn call $DeleteUpstream(\langle 1,2\rangle, \langle 0,1\rangle)$, which replaces all the occurrences of the redundant node with its only child. More precisely, first the bag of upstream arcs of node $\langle 1,2\rangle$ is traversed, and then all the links from the parents are re-directed to $\langle 0,1\rangle$. Then, the disconnected node $\langle 1,2\rangle$ is deleted. The resulting MDD represents the state space $\mathcal{S} = \{(*,0,0,*), (0,\{1,2,3\},\{1,2,3\},*), (*,\{1,2,3\},\{1,2,3\},0)\}$ (cf. Fig. D.9).

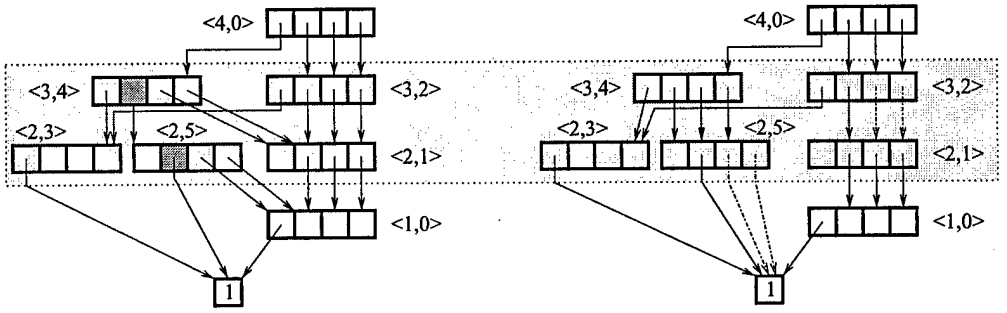

FIG. D.10. *Iteration* 3, *events* l_2 *and* l_3

Next to be examined are the local macro events l_2 and l_3. Both are enabled by a single node at the corresponding level. As a result, the first links of nodes $\langle 2,5\rangle$ and $\langle 3,4\rangle$ are copied in the last two locations of the array of downstream

pointers. The resulting MDD represents state space $\mathcal{S} = \{(*, 0, 0, *), (0, \{1, 2, 3\}, \{1, 2, 3\}, *), (*, \{1, 2, 3\}, \{1, 2, 3\}, 0)\}$ (cf. Fig. D.10). Event s_1 is then enabled by the sequences $\langle 3, 2\rangle \xrightarrow{3} \langle 2, 1\rangle \xrightarrow{3} \langle 1, 0\rangle \xrightarrow{0} \langle 0, 1\rangle$ and $\langle 3, 4\rangle \xrightarrow{3} \langle 2, 5\rangle \xrightarrow{3,*} \langle 0, 1\rangle$. However, no new states need to be added, since the outcomes $(3, 0, 0, 1)$ and $(0, 0, 0, 1)$, respectively, are already encoded in $\mathcal{S}$. Thus, the MDD remains unchanged.

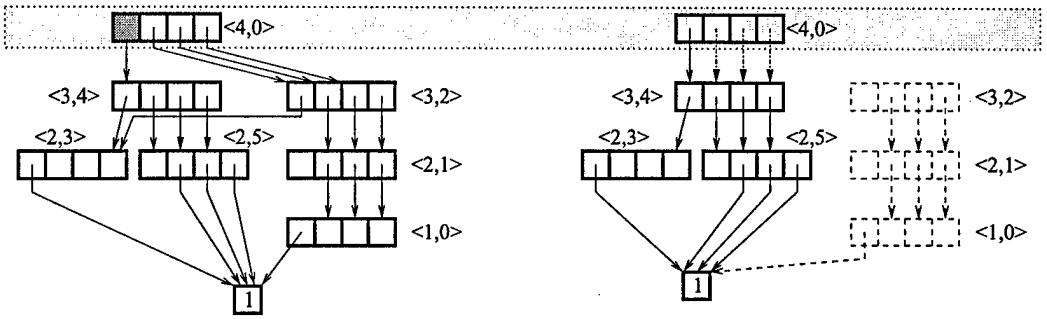

FIG. D.11. *Iteration* 3, *event* l_4

Finally, local event l_4 can fire again from node $\langle 4, 0\rangle$. The last three of its downstream pointers are replaced by the union of nodes $\langle 3, 2\rangle$ and $\langle 3, 4\rangle$, which is node $\langle 3, 4\rangle$, since the sub-MDD rooted at node $\langle 3, 4\rangle$ encodes a superset of the set encoded by the sub-MDD rooted at node $\langle 3, 2\rangle$. With all of its upstream arcs removed from its bag, node $\langle 3, 2\rangle$ becomes disconnected and is deleted along with its descendants by a call of routine *DeleteDownstream*. The purged MDD now stores the state space $\mathcal{S} = \{(*, 0, 0, *), (*, \{1, 2, 3\}, \{1, 2, 3\}, *)\}$, and the third iteration is finished (cf. Fig. D.11).

D.5. Final State Space. In the fourth iteration, no new reachable states are detected. Hence, the algorithm terminates and returns the root to the MDD representing the final state space. The final MDD is depicted in Fig. D.12. Note that level 1 is empty; the dotted node is redundant.

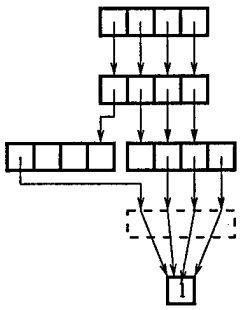

FIG. D.12. *Final MDD representing the complete reachable state space*